8-2016

EventFlowSlicer: A Goal-based Test Case Generation Strategy for Graphical User Interfaces

Jonathan Saddler
University of Nebraska-Lincoln, jsaddle@cse.unl.edu

EVENTFLOWSLICER: A GOAL-BASED TEST CASE GENERATION STRATEGY FOR GRAPHICAL USER INTERFACES

by

Jonathan Saddler

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfillment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Myra Cohen

Lincoln, Nebraska

August, 2016

EVENTFLOWSLICER: A GOAL-BASED TEST CASE GENERATION STRATEGY FOR GRAPHICAL USER INTERFACES

Jonathan Saddler, M.S.

University of Nebraska, 2016

Advisor: Myra Cohen

Automated test generation techniques for graphical user interfaces include model-based approaches that generate tests from a graph or state machine model of the interface, capture-replay methods that require the user to specify and demonstrate each test case individually, and modeling-language approaches that provide templates for abstract test cases. There has been little work, however, in automated goal-based testing, where the goal is a realistic user task, a function, or an abstract behavior. Recent work in human performance regression testing (HPRT) has shown that there is a need for generating multiple test cases that execute the same user task in different ways, however that work is limited in that it lacks efficient test generation techniques and only a single type of goal has been considered.

In this thesis we expand the notion of goal based interface testing to generate tests for a variety of goals. We develop a direct test generation technique, EventFlowSlicer, that is more efficient than that used in human performance regression testing, reducing run times by 92.5% on average for test suites between 9 to 26 steps and 63.1% across all test suites. EventFlowSlicer generates test cases for additional types of abstract goals beyond those used in HPRT and contains new logical constraints to support those.

Our evaluation on 21 realistic tasks shows that EventFlowSlicer can generate test cases based on a user goal, and that the number generated for each goal is non-trivial,

more than can be easily captured manually. It generates on average 38 and as many as 200 test cases which all achieve the same goal for a specified task.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Myra Cohen, for her attention and dedication to the success of this work and all related contributions we have made to the body of research in my field. There are many things I have learned throughout this process about the scientific method and software engineering research in general that have profoundly impacted my style of writing and my work ethic, and I appreciate the patience and trust of my advisor and my advising committee who helped me write this thesis, in this ongoing process of improvement and maturity.

My colleagues in the ESQuaReD lab, I wish to thank you for an excellent job at instructing me throughout the process of getting the work done, and getting it done right. To Wayne, Brady, Eric, David, Supat, Matias, and whomever else was in or near the room during my many spats, appreciation goes all around.

My family, for all your hope, faith, and wishes extended across many geographical barriers, I extend to you my utmost appreciation. I thank you for your trust in me, your trust that I could attempt and complete such a large task and gain a mark of such high prestige in my life. Family, both in blood, and in spirit, you know who you are – thank you for your support.

This research was supported in part by the National Science Foundation awards CCF-1161767 and CNS-1205472.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Passions to develop software drive different goals of what a software developer wants a product to accomplish. Often this is achieved with only a casual blueprint and the lack of formal specifications. Yet systems must still work without failing and the user should be able to perform tasks that are intuitive for him or her. Failures in software have been shown to cost the US Economy up to 5.9 Billion dollars annually [1], while the lack of an efficient user interface can harm the quality and negatively impact of most systems [2, 3]. And so these systems must be tested to validate correctness [4] and usability [2], and techniques should be developed to provide workflow guidance for users [5]. To achieve this we can generate and run cases to exercise the system from its interface.

Many of our systems today use graphical user interfaces (GUIs) as their front ends. In these systems the user iteratively interacts with elements of the interface to perform a task, which consists of a sequence of events to achieve some end-goal. For instance, to open a new file in most editors the user will click on the file menu item, select the open menu choice and then browse for the specified file, eventually double clicking on the file that they wish to open. Often, the same task may be performed in

multiple ways. Instead of opening the file using a menu, there may be a button that provides a single click to open the browse file widget, or the user may be able to type in a file name as a shortcut instead. Each variant of this task can be thought of as one way of achieving a user goal (an end state in the application achieved through a series of steps). Goals represent behaviors of a typical user and are the basis for system validation, usability evaluation and for providing workflow help for users who want to navigate the system.

While goals are often functional (e.g. open file), *how* this functionality is achieved may vary. For instance, in recent work on human performance regression testing, Swearngin et al. found that there were 81 ways to enter text, make it bold, and then center it in a common word processor using different combinations of buttons, menu items and keyboard shortcuts. LibreOffice [6]. For that particular goal the functionality of each of these tasks is the same, however how each different method to achieve that goal differs slightly. Goals may also be more abstract. For instance, a goal might be to change the appearance of a text box and the exact change may be undefined. In this case any method that changes the end-state of the text box satisfies this goal.

While there has been a large body of work on software testing (and test case generation) for GUIs [7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17], there has been little work for generating test cases that target specific user goals. Techniques for testing include manually capturing and replaying test cases to satisfy some use case, however in the situation described above it would be almost impossible for a user to identify and define all 81 test cases for a single task. Random generation is also a common technique, however this will not achieve a particular goal. Systematic testing techniques (our focus here) are often called model-based and these fall into different categories that we discuss later, graph or state-based techniques and domain language based techniques

that utilize customized languages and patterns to generate tests. However, none of these techniques can generate goal-based test cases as we describe in this thesis.

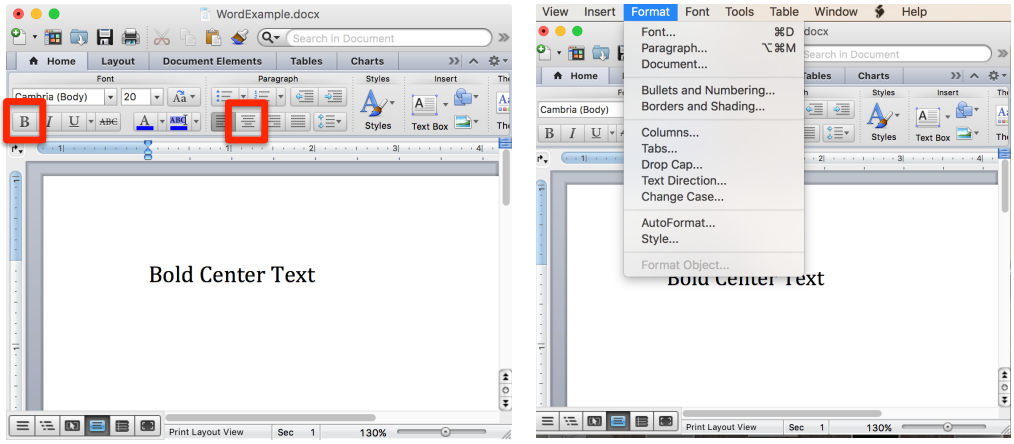


Figure 1.1: Microsoft Word Menus versus Toolbars

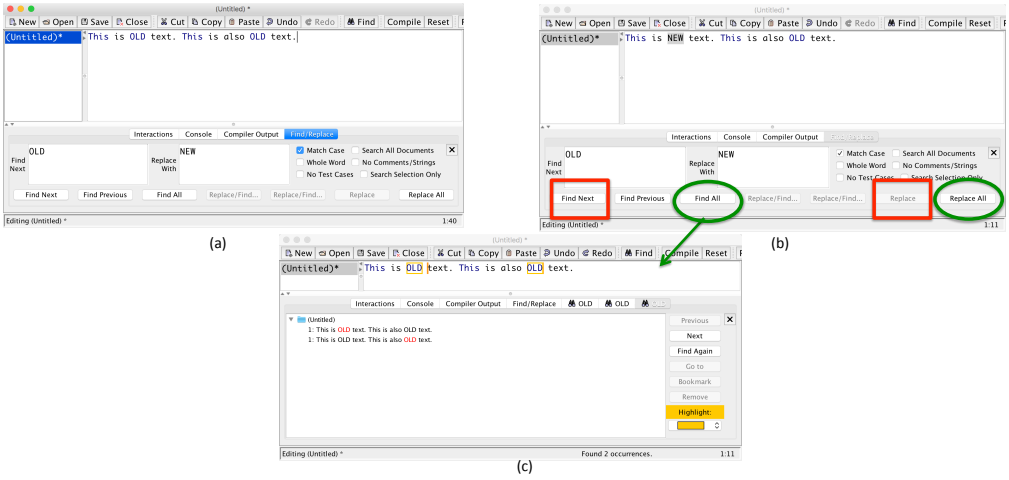


Figure 1.2: Dr. Java Search and Replace Options

1.1 Motivating Example

We begin by showing a few examples of the types of goals that we might encounter in a GUI application. Figure 1.1 shows the interface of the MS Word application. Suppose we want to bold and center some text. This can be achieved either by clicking buttons

(left side of the figure) or by using menus (right side). Of course the same task can also be achieved by a combination of menus and toolbars. This type of task is a concrete, functional task, however if this was a use case for a functional test case generator, one test case that achieves this goal would satisfy the use case. We call the different tests in this task, *structurally different*, meaning that they achieve their goal using different structural elements of the interface (buttons versus menus). In the work of Swearngin et al. [2, 6, 18] the notion of generating test cases for human performance showed that indeed there is a need to generate test cases that perform this task in all possible ways. This is because user interface designers need to understand how long it takes a user to perform each task and which elements of the interface are more/less efficient. If adding buttons to the interface, provides paths that increase the time it takes a user to make text bold and centered, then this may mean that their application has declined in quality and competitors may instead select other editors. In the case of safety-critical applications (such as a cockpit screen) the time taken to perform a task can mean success or failure of a life-saving maneuver [6].

Swearngin et al. [6] developed a technique to generate test cases for this type of scenario, however they did not directly generate tests for a goal – instead they generate all possible test cases that use the same sets of widgets that the task defines and constraints to remove tests that do not actually perform the given task. While this has a similar effect as the test case generator presented in this thesis, their generator does not scale (we show this comparison in our study section). Prior work using AI planning by Memon et al. [19] performed goal-based testing, but did not consider structural differences when generating tests (e.g. fewer test cases would be generated from that technique).

Figure 1.2 shows another example of a user goal. In this scenario we see the search and replace function within the Dr. Java Editor. In this screen there are multiple

ways to find and replace text. On the left part of this figure you see that we can enter text into the box to find and replace a word. We can select specific options (such as matching case) or not, and on the right we can find and replace multiple instances by selecting the find and replace multiple times or by simply clicking *Find all*. While also structurally different, we believe that the tests generated for this task amount to very different sets of task, even though the final state will be the same. We view these as having the same *Functional Goal*, but different steps are performed in different ways to get there. We differentiate this difference from the prior type of task which differed only in structure.

The last type of task that we identify in this work are those which have *Abstract Goals*. This type of test case is loosely defined. We might say that we want to generate tests to *change the background* but do not have a complete specification for what that means. In this case the user can change the background color, pattern, transparency, etc. All achieve the same goal, but are very different.

Our audience for this thesis includes software testers tasked with testing GUI interfaces, who may or may not be familiar with the fact that every interface has a state, (formally named "the preconditions and effects" in the work of Memon et. al.), in which properties exist that describe what the interface looks like and how changes have manifested. Measures of how this state is manipulated during a test case is what state of the art GUI testing has focused on within much of today's literature. In respect to the trends of the literature, our generator helps to solve the following goals regarding how test cases transform the state of the application:

As the tester plans their strategy to use a generator, a goal in mind for the use of this generator may be to first categorize a set of "wanted" or "expected" test cases into categories*, and then look to generate test cases using our generator that create all possible test cases to achieve this task*

We support the following types of tasks and for each genearte all possible ways to perform that task.

Structural Differences Only Different test cases per category are differentiated by only the type of widgets used to access each test step in the interface. All steps in one category do the same thing functionally to perform the same steps to access the main goal from the initial state. Some test cases perform steps with, perhaps menus. Others perform steps with toolbars. What matters is that function of each step itself does not differ.

Same Functional Goal All test cases in a test suite of this type must lead from the same initial state to a single end state at the very least. However, in this type of test suite, the steps to get to that single state might differ greatly in both the amount of steps taken and the type of steps taken, given the route through the GUI used to get there. The steps to reach the end goal are not similar at all in their function, while the goals are.

Same Abstract Goal A tester's end goal for each test case might be defined more coarsely than what differentiates the state of two instances. Different test cases in this generated suite type all stem from the same initial state, but the point signifying the end of every test case will be based on a more abstract definition of how the interface state *should change* in the tester's overall goal for the test case. The initial state and the final state are not equivalent of course, but the abstract goal the tester has in mind is still met via every single test case due to some change in state.

1.2 Contributions of this Thesis

In this thesis we present *EventFlowSlicer*, a test case generation strategy for GUIs that generates test cases based on user goals which are defined as tasks. We support three types of goals moving from the highest (structural differences) to the lowest (abstract goals) precision. Generation is achieved through a combination of user input to select events involved in the task and the definition of task constraints. All possible test cases (methods) are then generated that satisfy the goal and its constraints. We present a feasibility study followed by an empirical study in this thesis to show that (1) Our test case generation technique performs as well as that of Swearngin et al. with respect to effectiveness (we generate the same sets of test cases) however, it is more efficient reducing the time for generation by as much as 95% and allowing scalability to longer tasks (2) We can also generate test cases for a set of tasks that represent real user goals obtained from an online forum. We see that to generate all tasks for these goals is non-trivial (and not feasible by manual capture) since we have as many as 200 test cases in one task. In summary the contributions of this thesis is:

1. EventFlowSlicer, a technique for goal based testing in graphical user interfaces;
2. A case study showing that EventFlowSlicer is as effective, yet more efficient than another state of the art tool, can can handle more types of goal; and
3. An empirical study showing that EventFlowSlicer can generate a variety of tasks (satisfying all types of tasks we have defined) based on real user requests.

The remainder of this thesis is laid out as follows: In the next chapter we present background and related work. We follow this with a description of EventFlowSlicer in Chapter 3. We then present a feasibility study using EventFlowSlicer on a few tasks

in Chapter 5 and then follow this with a more formal empirical study in Chapter 6. We conclude and present future work in Chapter 7.

Chapter 2

Background and Related Work

We begin with some definitions that will be used in the rest of this work.

2.1 Definitions

Event – *A GUI widget on an interface paired with some associated action.* Widgets can be buttons, menu items, text boxes, containers, etc. Widgets on an interface are associated with one or more (possibly null) event handlers. A single widget can have more than one action, and each combination is considered a unique event.

State – *A collection of GUI widgets and their properties.* The state of a GUI is the complete set of widgets and their properties on the interface. States can get quite large and some elements of a state (such as the current date/time) may change dynamically, therefore some techniques will prune and define the essential state of an interface.

GUI test case – *A sequence of events that can be executed on an application.* An example of a test case is the sequence <Cut, Copy, Paste> where each of these

events has an associated widget (e.g. a cut button or paste menu item) and an event handler (e.g. double click, right click, etc.).

Step – *A single event (or group of events) within a test case that performs some function on the interface.* Steps can be a single event such as <Cut> or it can be a longer subsequence of the test case.

Task– *A definition of the user goal as it should be achieved on the interface* A user task is a way to achieve a particular goal on the interface. Usually there is more than one way to perform a given task such as "bold and center".

Method – *A single test case that performs a task.* Each task may have many test cases or methods that achieve the same goal.

2.2 An Overview of GUI Testing

GUI applications are more complicated than their CLI-based (command-line-interface-based) counterparts. Users in today's world expect GUI's to be responsive to many more actions than CLIs. Unlike CLI's, GUI's contain widgets that accept keyboard input and additionally buttons, menus and menu items. The widgets included in the interface must remain responsive and perform concrete tasks according to specification for the tool in order for it to be perceived to function properly. The actual functions in GUI's today often depend on the ordered sequences in which groups of events are activated. We describe several testing strategies next.

2.3 Model-Based Testing

Model-based testing provides a systematic way for GUI testers to find bugs. It uses a **model**. A model represents the interface as a graph or state machine (or in some other abstract representation). The model is then used to define what elements of the interface should be covered during testing and can be used for test case generation (generation of sequences of events). The primary model-based testing strategy we will focus on in this thesis is called **reverse engineering-based (RE-based)** GUI testing, in which the interface can be reverse engineered by executing the application and observing which events are found on the interface and which events can follow others. However, models are built primarily for automated test case generators and do not have a human-based goal (as we require).

2.3.1 RE-based Testing

RE-based GUI testing is a fairly new technology, gaining its strongest popularity as an alternative to manual GUI testing. RE-testing has historical roots in model-based testing. White et al. introduced the concept of generating test cases from interactions with the GUI, strung together to form "complete interaction sequences (CIS's)." The concept of the CIS can be used to establish a modelling of the reliability of how well the interface works. One can test combinations and interlacings of various CIS's to expose states of the application, be they *"expected"*, or *"surprise"* or *"defect"* states. State based testing the way defined by White et al. has been replicated in many forms, especially in much of the work done in RE testing. Memon et al. pioneered the concept of reverse engineering tests on an interface [4, 10, 11, 14, 20, 21]. Memon applies testing of GUI oracles - using "GUI state" and "GUI event model coverage" as a guide. In his words from [10] he reduced his scope to include the following definition

of GUI's that are testable:

"A GUI is a hierarchical, graphical front-end to a software system that accepts as input user-generated and system-generated events and produces deterministic graphical output. A GUI contains graphical *objects*; each object has a fixed set of *properties*. At any time during the execution of the GUI, these properties have discrete values, the set of which constitutes the state of the GUI."

Memon et al. refers to a subset of all GUI interfaces available in today's market. In summary, a GUI is at any given time: A) A pair of two sets: *graphical objects* and B) the *properties* of those objects made visible or kept invisible by the system, from which can be derived C) the *state*, or an account of all discrete values of the interface's properties. Given our conditions of a GUI are met, then according to Memon et al.'s specifications, we can begin testing by attaching preconditions and effects to interactions: the *effects* change the state after each **event** or action on a widget fully executes, and the *preconditions* define what of the complete set of events available may be executed at any given time. The key takeaway of Memon's work on fundamentals of testing GUI's this way: as we measure the effects of actions on GUI's, the events we carry out on its objects, we test the application.

2.3.2 The Event Flow Model

Memon et al. also found that if we keep track of some things but not others we can use an abstraction of preconditions and events to over-approximate all possible interactions with the interface. Memon developed the concept of the *event flow model*, a stateless

(effectless) model of interface interaction, created by tracking the preconditions in which events may be executed on an actively running interface, but not tracking any further information about the effects [10]. Memon et al. also developed techniques to dynamically extract and EFG with little user input and created a testing framework called GUITAR [22] [7]. GUITAR dynamically reverse engineers interfaces using a breadth first search as it automatically gains information about GUI properties, events, and preconditions, and inserts this information into persistent testing assets called "model files" that are designed to be used together as a basis to develop test cases and automate their execution. The events for each test case come from the Event Flow Graph (EFG) file, while the GUI forest (GUI) file helps gather enough information about the structure of the interface to help the test case execution tool (called the "replayer") automate each test case. GUITAR has gained popularity as a prominent tool in RE testing, and has several great advantages over previous model generation strategies due to its ability to find, automate and test thousands of GUI test cases at a time.

2.3.3 Domain Specification Language Based Testing

Our mission is to identify what kinds of testing tools aim to solve problems related to user goals, or tools that locate instances where bugs are preventing users from achieving an end to intended tasks.

A form of tester-specified "natural-language based" GUI testing came to the forefront in the literature. This is still an up-and-coming strategy in which testers use language(s) recognized and constrained to the domain of testing to specify tests. This field of languages are fitly called domain-specific languages. This field of testing is fitly called domain specific language-based (DSL-based) testing. DSL based testing is highly

reliant on the use of models to generate tests [23], and thus is a modelled-generation strategy. In **DSL-based testing**, testers use the DSL chosen by the model creator to specify the model, and then reuse components of the DSL to specify the tests themselves. DSL tests are known to be highly configurable.

The takeaway from the impact DSL-based testing has had on GUI testing is that testers are given the ability to define directly the kinds of things in need of testing. In practice this has shown to be effective when design of the language is allowed to take place from the ground up [24]. The test cases that back certain goals users have in mind when using a software application are generated much quicker.

Work by Paiva [12] [25] [26] has gained attention. She noted in her work that event flow models are hard to change, they are hard to break down and modify to meet the needs testers have in exacting specific functionalities out of the system. She advocated for a modelled-testing strategy that received more input from the tester on what oracles needed to be tested. Work by Bae et al. [13] shows further evidence that raw RE testing on large interfaces may not be as effective as other efforts to test GUI's that go directly after events in the interface themselves, efforts which lie outside of the realm of model-based testing.

However, for a DSL-based tool to be effective, it requires that a developer be well versed in how to specify a model [25]. No matter what GUI tool we end up using, the tool still needs to know how to automate the tests, and thus needs directions on how to interface with the system or the system's replacement. "Operators" in DSL-based testing are what help the tool that's doing the modeling interface with the system [12]. It is up to the developer to provide these and extend these when needed when more features are added. EFG based systems come with operators supported by default: testers nor developers need to define these to get the modelled-generation automator to generate tests or even to get it to run, because these operators are

reverse-engineered from the runtime state of the interface. There is another key with DSL's. They require that the creates the cases where faults may lie in the interface. Hence the lack of a model may bre problematic.

2.3.4 RE-testing with a Focus on Goals

Swearngin et al. [6] proposed a body of work that focused on goals users with respect to the efficiency of performing tasks on an interface (human performance testing). Their goal was to generate tasks that an expert users performs. Their work is undergirded by the fundamentals of RE-testing, drawing from knowledge on how GUITAR measures state using RE techniques. Unlike GUITAR however, the Human Performance Regression Testing (HPRT) generator focuses on generating results that denote tasks ending in a finite goal state. As for the basis, the tool generates test cases primarily based on user defined test cases that the tester can capture on the real interface with keystrokes and mouse clicks, not a set of GUI structure or event flow assets like GUITAR, and not (solely) based on GUI state. A string of information about the events the user executed on the interface is saved and set aside for future modeling as events are captured. Multiple strings of events can be saved and set aside as separate concrete "methods" for achieving the same "task." Having completed the user capture phase, model generator reopens the application on its own and replays each *method* recorded by the tester on the real interface. The generator adds a bit of state from the application back into the process, but at a time only after the user-captured tests have been generated. HPRT, having gotten snapshots of states from its replay phase, does not use state to determine when a test case has failed as in older modelling-generation strategies, but instead uses state to determine whether new states have been entered *during* the task the user captured.

Constructs built into the generation technique then allow for the generator (now in the true *modelling* phase) to find and denote strategies to reach the end goal that the tester initially did not input to the program, and present them to the tester as added test cases in the test suite. This is a huge advantage of this generator unforeseen in the field. The HPRT generator can gather such information when multiple event strings with crossing paths through similar states are captured and sent to the generator for modelling. The study performed by Swearngin and Cohen primarily focuses on how tasks become less or more efficient as the development of an interface advances out of an older state to a new state. Swearngin also completed work on a second test case generator based on GUITAR model assets used for modelling test cases using **constraint-based** techniques [18]. We'll cover this generator much more in future sections.

2.4 How to RE-Test an Interface

To show how to test an interface using RE-Testing, we are going to walk through the operation procedure of perhaps the most popular tool in the software testing research community available, *GUITAR*, which stands for the "GUI Testing Architecture." The steps used to generate test cases using GUITAR are listed next. These steps are very similar to the process we use to operate our generator. We will now explain the details behind each step.

Rip Capture a snapshot of all the GUI objects visible on the interface and their hierarchy and their visible/invisible children. Each GUI object are captured in the snapshot is called a **widget**.

Model Conversion Convert the output of the rip into an event flow graph.

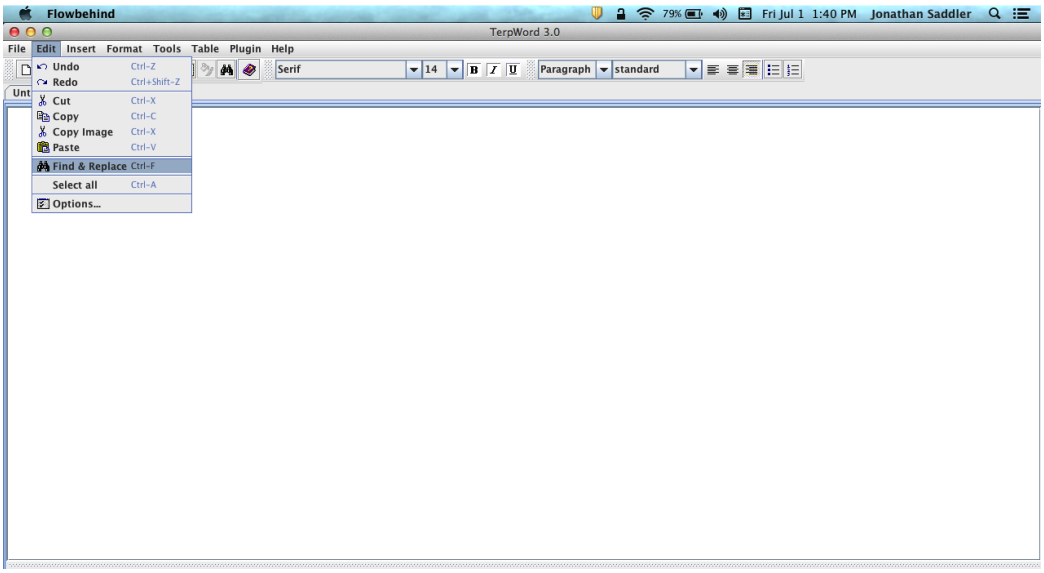
Generation Generate test cases from paths that flow through items in the event flow graph.

Replay Automate the execution of each test case using the application under test.

During this process the application is open and actively running, while subprocedures running alongside the application probe the running code to manipulate the interface in an algorithmic fashion, presenting a rather vibrant visual of the application "unfolding" at each step in the process.

GUITAR's ripping algorithms are generic enough to recognize differences between how applications running under two different frameworks must be traversed for information, ensuring that we gather and store properties and other information common across all GUI's. GUITAR accomplishes this using its modular architecture, allowing the same tool to connect to and manipulate applications running under separate platforms of interest using **platform plugins**. Platform plugins specialized for a platform insert code into the algorithm at specific places where platform API must be called to obtain GUI properties. While it is optional to customize the model generator and test case generation in other steps with custom platform-based plugins, selecting a platform plugin built for the platform the application under test uses is required when dealing with the rip and replay steps.

GUITAR starts from the root components of a GUI, which is by definition hierarchical, and works its way down from the root (typically an application "window"), to the components it can find which are visible within the root. For components that might expand to contain hidden children (such as menus), the algorithm attempts to expand the container, treating the container as the new root, and repeats the process to fully exhaust the contents of the container's items, before moving to the next sibling it finds adjacent.



The GUI Ripping algorithm is very akin to a typical Depth-First-Search algorithm, being run on a real GUI as a basis. Using this process, the order in which GUITAR finds information about how widgets are nested and exposed to the user, also helps to expose a hierarchy in how elements may be made legally accessible to the user. This information about hierarchy stemming down from the main window of the application is quite useful, and is the first list of facts that gets stored in the output, a model asset called the "GUI Forest." It is called a forest not a tree, because the end of this list of facts will also contain other rooted trees GUITAR finds while ripping buttons that expose new windows.

We use JFC Guitar as a backend for our new tool, a platform plugin that comes standard with GUITAR that is built to reverse engineer applications that rely upon GUI components from the Java Foundation Classes (the Java Swing/AWT) platform.

2.4.1 Model Conversion: Derive an Event Flow Graph from the Structural Model

Stored in the GUI model are strings of information about the interface's hierarchical structure. As data from the GUI model is read in by the model converter, it is queried for widgets that contain properties useful for event execution. The generator can determine from the information provided where widgets fall in the hierarchy, whether widgets are hidden by their parents, and what kinds of actions we may invoke on the widget to activate its primary function or functions. Since we are not just ripping buttons, but also text boxes and widgets requiring more than one type of input to fully operate (table cells, select-from lists, and combo boxes), this information about widget operation is essential for test case generation and will be stored in the EFG file as each event is queried. it can also tell the size of the widget and the location of where each widget can be located (relative to the parent container). Such information is kept out of the EFG file, and referenced by the test case generator only when needed, since these are properties of state, unrelated to the execution of the test case steps applicable to the widget.

Given such rich information about the flow of control needed to legally access widgets, GUITAR gives access to a **flow graph**, capturing a stateless over approximation about how the interface may be legally *traversed* since the last rip took place. Since this depiction is a stateless over approximation, certain widgets (such as ones that are disabled) may in reality be inaccessible to the user upon startup, though flow may pass through these widgets to other widgets in the EFG, as if they were.

The standard *EFG converter* GUITAR provides creates a one-to-one association between each widget found in the GUI file that is likely to respond to peripheral input from the user, and one event in the generated event flow graph. As for the edges, we

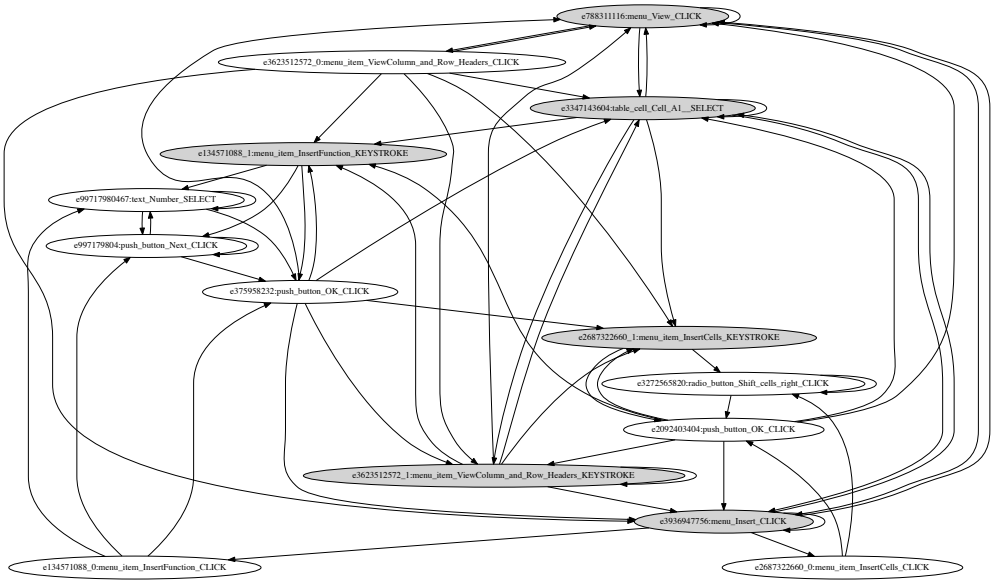


Figure 2.1: EFG depicting Absolute Value MK task.

can see from the example that an edge always reaches from the parent of every widget to the widget itself (or if the widget resides in the main window, then a self edge is created), and an edge going to and from every two widgets that coexist together simultaneously on the open interface.

An event flow graph is depicted in figure 2.1. In this depiction, shaded-nodes are the nodes discovered first by the rip called “initial nodes”, accessible upon application startup from the application’s main window. Edges leaving gray nodes lead to some events that are accessible only from within subcontainers of the main window, and thus only legally accessible after initial input from the user. This particular EFG depicts the absolute value task EFG, related to a spreadsheet application. The nodes depicted re-enact what is required to perform the subtasks of the goal, which include inserting data into a table cell and shifting data one cell to the right. We can see in the bottom right corner in this depiction, white unshaded-nodes adjacent to parent

events that open the "Insert Cells" menu, namely a keystroke event, and a menu item event. The white unshaded represent events that are only accessible from within subcontainers of the main window, hence depicted in the figure is that it is necessary to open the Insert Menu before confirming the option to shift cells to the right and click OK. Likewise, in the top left corner, it is implied that the "Insert Function" wizard must be first opened, to give access to the [Number text box] and the [OK button] used to confirm and insert the function into the cell. The [Next button] is also available throughout this substep, but upon further inspection, this button is actually disabled after it is clicked the first time.

2.4.2 Generation

GUITAR contains several test case generators and the tool can be extended to add more. By default, GUITAR's built in generator is designed to traverse the EFG in a manner congruent to how the ripper traverses GUI's. It starts from the top, an initial node in the EFG, and works its way down until stopping criteria are reached, and then moves on to other initial nodes. The user can select the length of test cases (i.e. generate all length 2 test cases), or a random selection (generate X random tests of length N). Note that the random generator actually does an exhaustive generation at the given length and then randomly selects a subset for the user. This limits scalability.

2.4.3 Replay

In the end, we can automate the execution of each test case. The GUITAR replayer features both the use of a platform-plugin used to interface with widgets, and also behavior plugins that can be used to capture information about application state such

as code coverage and properties of certain widgets of interest. If any test case fails to execute, due to a change between rips in how the application works, one can return to the initial step, generate a new rip and EFG for the application, and create a suite of new test cases.

2.4.4 Summary of RE-Testing

The GUITAR tool implements all the algorithms that reverse engineer information from interfaces to generate tests and walks the tester through each step of using their strategy to generate tests. With the stock replayer found in GUITAR, many many test cases are generated and replayed based on whatever information is supplied to the EFG through the ripping process, the size of the suite being on the order of magnitude equivalent to the amount of edges present in the EFG and number of widgets we pull from the rip and the GUI asset to create it. In short, with no configuration and stock plugins, every possible test case of a given length that can be generated on the entire interface is outputted as a test case and is immediately replayable upon relaunch of the application via the replayer. The tester can observe the state records outputted by the replayer, or view the outputs recorded by replayer behavior plugins, to detect bugs.

2.5 Related Work

We explain work related to the current state of GUI testing. To help group together content across the field by make two cases for the need to improve the state of the art of GUI testing, and how GUI testers aim to solve these problems. Hereby, we define a basis for why new GUI test case generators are needed. In the final section are key points describing the body of research being done in GUI testing currently, and how

this generator proposes or doesn't propose state of the art technological advances in the areas covered by other technologies.

2.5.1 Case 1: Time Spent Computing Test Cases

Model-based generators generate and automate test cases that exercise the interface in an exhaustive manner, so as to thwart possible bugs in a matter of time that a human cannot feasibly. This has proved to serve very important roles in a variety of domains. Smartphone model-based generators, for instance, have been written to test suspicious executions of code as the user navigates through the interface, as simply clicking the buttons of a smartphone can lead to information being leaked to unknown remote sources [27]. Model generation stops researchers from having to manually navigate interfaces to test them saving resources on the time and personell necessary to test the application. [10] [3] [28] [6].

Memon brings up the fact in a seminal paper on EFG flow models, that models constructed for the model based generator can be used and reused to generate multiple test suites of varying qualities on an application [10], but he fairly pointed out the caveats in work earlier than and leading up to his, regarding the amount of time it takes to get back test cases that can be used. Early iterations of these "event flow model"-based generators suffered from a problem in order to get test cases that would run: test suites were based on *"modellette"* miniature models of events themselves, called **operators**, which needed to be hand-coded by humans in order to get the replayer up and running. The replayer would use these operators to actually press the buttons on the interface, and interact with the software. The process involved a model checking process to verify that the software correctly responded to operator input. The process of building these models added hours and even days to the process

of generating tests, and was considered to take up in the words of the author "a significant portion of the time" spent on the overall testing process.

In later envisionings of model-based testing, the time taken to create useful test cases has gone down, but in reality the time has shifted from operator coding to other aspects of the generation, most importantly the time taken to compute the test case sequences to be outputted. The generator cited in the seminal paper by Memon, indeed, took a minimum of 100 and a maximum of 820 seconds (aprrox 13 minutes) to completely generate a test suite for an applicaiton, not including the weeks that it was reported to have taken to encode and test operators.

Memon and Nguyen later improved this process in an implementation of a new tool implementing much of his work, GUITAR [22], a tool that could reverse-engineer interface operators rather than requiring the need for humans to hand code them. In studies performed on the performance on this new tool, the generator was tasked with generating thousands of test cases to uncover seeded faults. Memon and Nguyen reported their technique to be quite successful. The process taken to generate test cases and execute them however, still took a long time.

Other authors of similar tools followed suit. The authors of another test case generation tool AutoBlackTest [17], in a comparison study, cited that GUITAR takes a between a hard minimum 6 and an average of 10 hours to completely generate test cases of length 5, while reporting 12 hours for its own tool to run. EXSYST, a tool that uses a finite state machine to model test suites, and that creates and evolves test suites using a genetic algorithm, reported their tool takes 15 minutes to completely generate test suites. [9]

If a tester doesn't have hours to days at their expense left in the budget for both creating a test suite and exercising simple functions, modern state-of-the-art test generation techniques may not measure up to be suitable.

2.5.2 Case 2: Test Cases Generated Are Irrelevant to the Goal

Memon also brings up in his seminal paper that the tests, while having some qualities, might not possess all the qualities a test case needs to be useful.

Memon hypothesized in [10]:

"On one hand, since a human tester is involved in creating test cases with capture/replay tools, the tester can make intelligent choices, resulting in effective test cases. On the other hand, an automated technique has the advantage of being comprehensive, in that requirements such as 'cover every event with at least five test cases' can be implemented into the [event-space exploration sequence] algorithms."

This is an important point made in a paper supporting many modern ideas behind model-based testing, and moreover a statement which is essential to the cause behind our work and the work of many researchers in the testing community ***to date***. It is true that exhaustively generating models through the use of small-length **smoke testing** will cover all possible combinations of events length at most "x", and will exhaustively test most widgets within the interface. While smoke testing on an interface can exercise many faults based on small-length sequences, such testing can create non-sensible sequences.

In [6], Swearngin et. al. point out that typically not all test cases generated using a model-based approach serve useful afterward from a usability tester's perspective. True, in the culture developed around software usability in applications, GUI's that involve using say toggle buttons to turn modes on and off do not typically require

that such a button be repeatedly mouse-clicked to engage the key function. Thus smoke tests involving two or three presses of a "boldface this text" button hovering over a document opened in a rich-text editor, won't meet needs for a tester aiming to test whether the feature works. Neither will a case involving three or four entries of the same text into a text field help a tester decide whether the submit button for that form field works. However, these are the kinds of test cases we get back from a model-based generator, unconstrained to generate whatever test cases it wants.

The problem with the output, however, exercising not enough functionality in an interface or exercising the same functionality repeatedly is a double-edged sword. Test cases need to be long enough an consist of just the right amount of steps, or else **tasks** that require lots of interaction, yet that are still serve as a crucial part of the typical user's normal workflow, will never show up in the output. Thus another drawback for unconstrained test suites is that such a strategy cannot feasibly test event sequences that long, or long enough to exercise goals that are "complex enough" to involve a wide variety of events executed in just the right sequence (and even repeatedly if necessary).

An outcome of smoke testing, DART [11], was used to generate test cases on applications in which faults were seeded into AUT source code. They found their method to be quite effective at finding faults. The authors were forthcoming about an observation they note while reviewing their efforts:

"Finally, there are events in the GUI that are enabled only after some other event sequence has been executed. If the required event sequence is longer than three events, the smoke test cases cannot execute the code associated with the disabled events."

Often test cases may involve a wide variety of unique events, say 9 or more, thwarting any attempt to run to completion algorithm attempting to exhaustively search an exponential number of paths.

We will explore this problem in greater depth. Some of the goals we aim to achieve using our test suites involve closely-tailored orders (Type search parameters in the "Search For:" and "Replace With:" text boxes before clicking "Find"). To generate many members of these test suites with 16 to 25 steps each as we demonstrate in our results, using state-of-the-art testing methods, would take infeasibly long amounts of time to do.

To date, we don't know of any other GUI testing technology that aims to compute test suites this long.

2.6 State-of-the-Art Technologies

Not all testing tools for GUI applications are based on flow models as ours is, and in fact use completely different techniques to detect problems in applications. *FlowFixer*, by Zhang, Lü, & Ernst, is a program-analysis based tool that recommends fixes based on an analysis of two source code versions of the same application, where GUI application features may have been removed in the second, or reimplemented in such a way where the older workflow no longer functions as normal. The FlowFixer tool simply recommends fixes to workflows. The FlowFixer tool, however, is not a replacement for a true test case generator, as it only recomments widgets that are potentially missing within broken test cases.

Mariani, et. al. in [17] focus on "ant-colony optimization" techniques, which are search techniques that generating relevant GUI test cases using reinforcement learning. They compared their tool, AutoBlackTest with Memon's *GUITAR*, and found the running time is comparable to GUITAR's in generating small test suites of length-5 and below. Carino et. al. [16](CarinoAntQ) also published a selection of tools "Ant System" and "AntQ" that utilize ant-colony optimization rather than models to

generate GUI-based test cases. Test cases done using this method take a long time to generate, and the algorithms are nondeterministic.

2.6.0.1 Domain Language Driven Techniques

Micallef did a study on the advantages of using Domain Specific Language based testing tools for testing GUI applications [24], specifically using a modelling-language testing suite called Gherkin [29]. The authors performed a real-world industry study, and found interesting findings when installing and studying an implementation of a domain specific language for a gaming company and international publishing house. They found that a direct benefit of using a modeling is that, after its initial design, a testing strategy using them can easily be imported into a team's workflow without interrupting team productivity. Another finding was that the sharing of a DSL's specifications from partner to partner across a testing sector can inadvertently "blow up" the testing language when strict standards on how and how not to extend the language aren't in effect. Thus the test suites built using DSL's can become redundant, a notable drawback for companies whose priorities may be focused on reducing the cost and thus the number of test cases to be executed before product release.

Interestingly Ana Paiva's *Pattern Based GUI Testing* tool (PBGT-Tool) is perhaps the model-based testing tool recently released to the community that most closely resembles ours in terms of purpose. Testers model their interfaces using patterns based on operators coded into the platform: the user simply uses a sequences of mouse clicks to indicate how the testing procedure will ensue, and will utilize constraints to bind those behaviors to real sequences, the most prominent constraint being "Sequence." PBGT-Tool does not use a reverse engineered event model, but instead PBGT testers can hand-code the model of the the interactive flow through the interface themselves in a modeling launguage built specifically for the PBGT tool called *PARADIGM*.

Models reverse-engineered from the interface are easier to create than hand-coded models. They are less prone to user error, and easy to quickly generate and reuse. PBGT-Tool is also a plugin reliant on other testing software, such as a constraint-format generator, Alcoa for its Alloy-format constraint inputs, and a runtime platform, Eclipse IDE, to fully operate. Our tool on the ohter hand is standalone. The user simply requires an updated version of the Java runtime (Java SE 7) and their application's compiled class files to create, constrain, and replay tests.

2.6.0.2 Model Based Strategies

Much work has been done on model-based test case generation, especially in the volatile arena of today in GUI testing. There are other generators besides ours that work from the basis of a model of the GUI interface to generate and replay test cases. *GUITAR*, our star of the show for most of this thesis so far, was published by Nguyen and Memon. Extensive work on extending the functionality and scope of problems that guitar can solve has been carried on in work done by Gao, including a recently released newly developed workflow-based generator called *testPatch*.

Like tools built into guitar itself, work on test patch focuses on "oracle-backed" testing, or testing of an interface with a focus on whether the effects produced by executions of replays on the interface produce a desired result according to an oracle. However, as reported in results from test suites generated on applications GUITAR has been used on, [17](MarianiABT) [10], test cases GUITAR generates tend to be quite small (often reported by studies are test cases containing 5 events or fewer).

Gross, Fraser, and Zeller authored *EXSYST*, a dynamic, program analysis tool deviating from the use of event flow models in favor of finite state machines. They use genetic programming to dynamically generate test cases for a software system or a GUI interface, namely ones that quickly point to faults in application, and conclude

that their idea lands the blame quickly on test cases that exercise real faults in the application. The key here is to generate test cases that give high code coverage, and ones that exercise the interface in a manner less likely to report test cases that are false positives not behind the cause cause of real application faults. Our generator wishes to return the entire scope of test cases covering a particular task, not only the ones that cause or don't cause failures. The tests returned by the algorithm, being a search-based algorithm with a random seed, are non-deterministic. Our algorithm, on the other hand, is deterministic.

Swearngin et al. devised the HPRT generator, a constraining test case generator that implements functionality in the closest similarity to ours, by allowing users to constrain every test case to follow "Required", "Exclusion", "Order", and "Repeat" rules. The HPRT generator, as it was dubbed, effectively reduces the amount of test suites we would retrieve from an unconstrained RE-model-based testing method, in order that we might gain test cases that are useful from a usability perspective.

Chapter 3

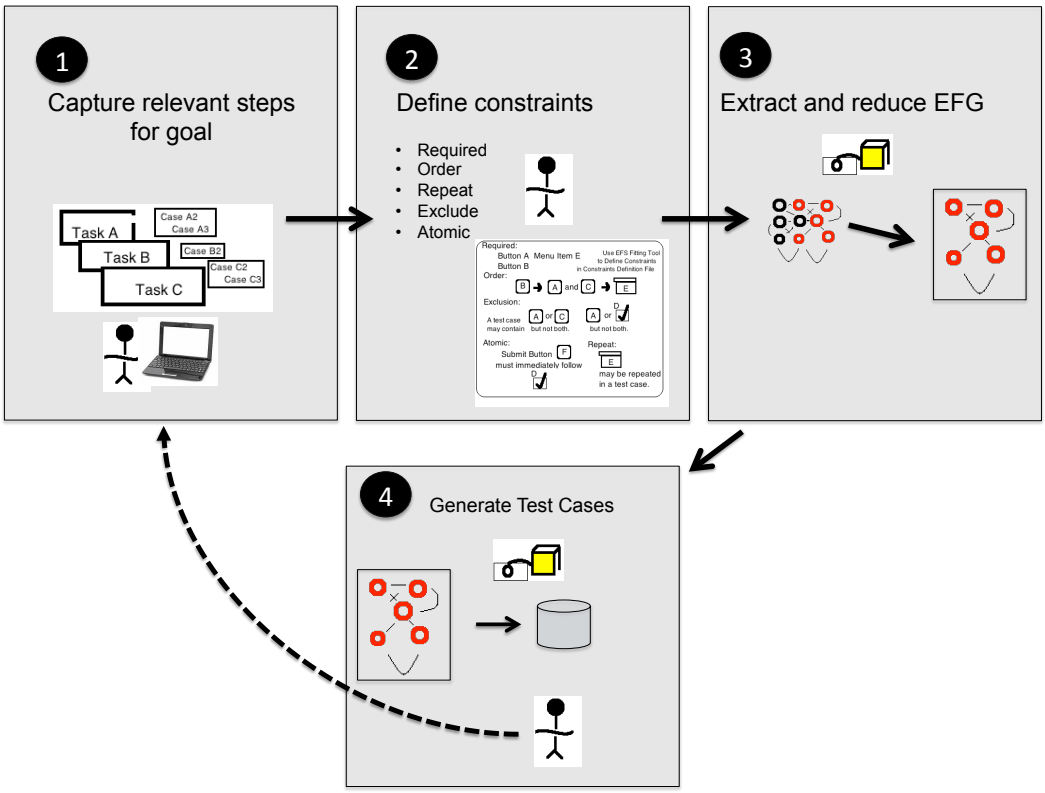
EventFlowSlicer

In this chapter we present EventFlowSlicer, our strategy for goal-based test case generation in GUIs. A tester or software developer needs to have a bit of world knowledge about how the application, and what features lie within it, are designed to be used. Testers are either trained how to work with the application themselves, or come to the table with such information, so the idea that this knowledge is available remains a given assumption throughout our process. With that information, our tester, Tim, could use EventFlowSlicer to test specific features of his application in many ways, use its constraints system to focus each test case on executing the goal that he is trying to perform, and perform a focused test case generation instance using only the widgets that are relavent to him.

3.1 An Overview

The EventFlowSlicer end-to-end generation strategy is a goal-based test case generation strategy. We use a 4-step process called EventFlowSlicer that helps generate test cases from an event-based flow model space, in which test cases not relevant to realistic

Figure 3.1: EventFlowSlicer



user behavior are brought under control both by general "given" global constraints imposed by the generation platform, and by rules the user can specify. Figure 3.1 shows the process which allows for (optional iteration) back to the first step since we have learned that the modeling sometimes is an iterative process.

In step 1, the tester captures relevant steps for a goal using a GUI interface, where each event captured will become a node in the EFG, and given space to define as an element of a constraint step 2. In step 2, the tester defines the constraints, selecting from widgets captured and placing them in the constraints groups, thus devising a logical plan on how widgets may or may not be implicated by events in certain test cases. In steps 3 and 4, a ripping process occurs and the interface is reverse engineered and then reduced using the events captured and the constraints. If the user is not

satisfied with the structure of the test cases, the tester can return to the beginning of the process to recapture new widgets and constraints, or can skip the first step and redesign constraints if the constraints defined don't require a larger or smaller scope of widgets. Finally, all test cases are generated from the reduced EFG given the set of constraints.

We provide a walkthrough of how to use our generator, accompanied by a running example brought in from a real-world use case of the editor *JEdit* we pulled from a question on StackOverflow. We give the reader a walkthrough that ties explanations on how this proposed generation platform works, together with expected inputs and outputs on a real-world instance to generate a test suite conceived in order to demonstrate and exercise as many aspects of the same task defined in the StackOverflow question as our generator can provide.

3.2 Step 1: Capture the Events

The first step of every test case generation process is to indicate what objects on the interface we wish to center our tests around. Only events pooled from the widgets we indicate to the system will appear in the outputted test suite. In this step, the way the tester indicates to the system which events should be included in the test suite is via "capture", via a point and click mechanism using a capture mechanism hosting part of the actual interface as a backdrop. This capture mechanism will identify those events that the user interfaced with (clicked or typed text into) back to the test case generation platform, making it possible to focus tests suites to be generated specifically referencing events the user feels are relevant.

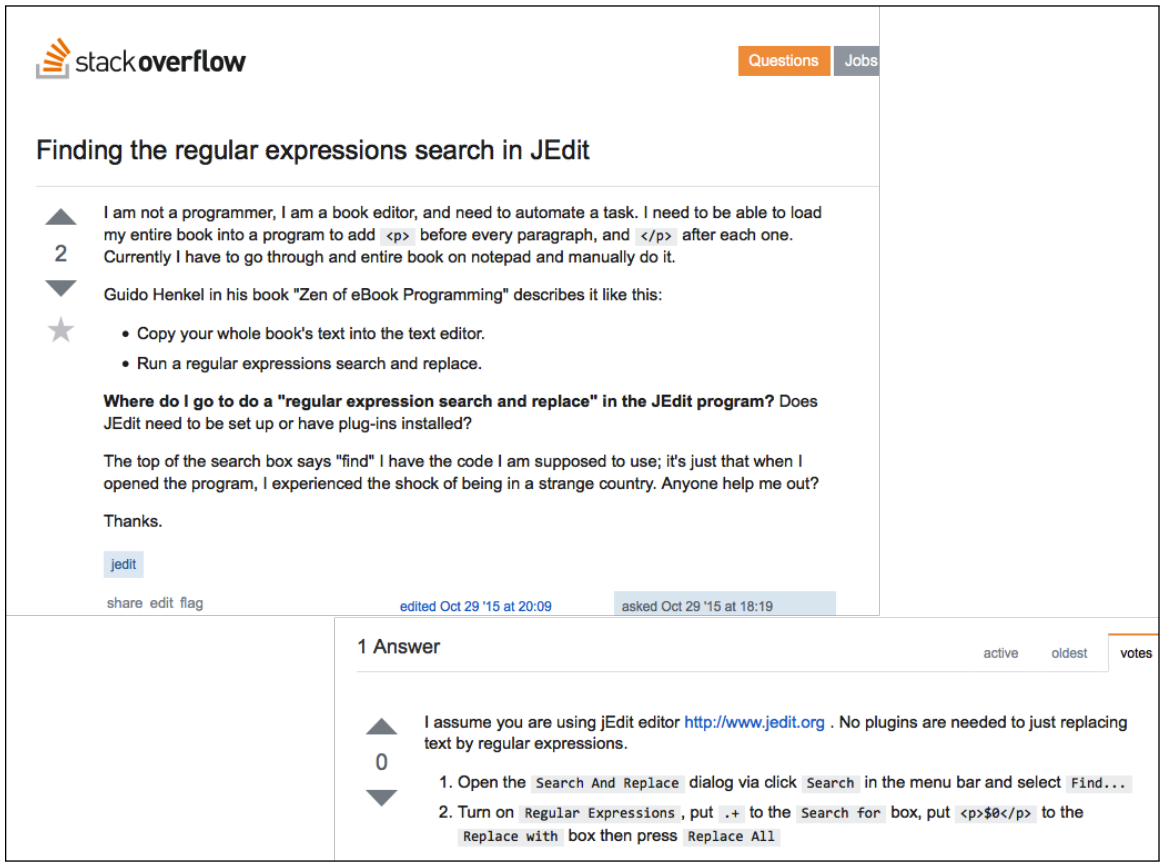


Figure 3.2: Snapshot of question and single response from users on StackOverflow.com

3.2.1 Running Example Introduction

To demonstrate motivation behind why generating test cases this way might be useful for testers in the real world, we took a question from the website "www.StackOverflow.com" and attempted to turn a question that a user asked into a question we could answer using our test case generation process.

The original poster refers to a need to figure out how to perform a task in the JEdit text editor, an open source text editor built in Java Swing that allows many customizable options for editing text.

3.2.1.1 The Definition of the Task

We'll define a goal for ourselves to establish the focus of this walkthrough example: The task will be

Quickly wrap four separate lined paragraphs in an opened text file with a " " tag, using only the "Search and Replace" text editing function (no HyperSearching or opening multiple buffers or extraneous JEdit search options), all using the JEdit Text Editor

For this real example of how the test case generation should proceed for the given task, we have precompiled a test .txt file according to a few specifications which (given the lack of an exact specification provided by the original poster of what a working file to modify with search and replace looks like) match what an input file that we could execute such a task on might look like. Specifically, our input simply contains four paragraphs separated by line breaks. We wish to wrap " " tags around every one of these paragraphs.

Put formally, our goal is to generate a test suite that exercises as many realistic, non-redundant behaviors on JEdit that bring the state of the application from its initial state, to a state that accomplishes the mission of wrapping every line in the input file with tags, using only the Search and Replace window built into JEdit's GUI interface.

We'll be using the JEdit text editor, version 5.1.0, as our application under test for this example.

3.2.2 Steps to Capture

Using the JEdit text editor, our hypothetical tester wants to carry out the task stated above by adhering to the following steps once opening an input .txt file containing the input we want to work on.

1. Locate the Search menu in the menubar at the top of the window, and click the "Search | Find..." menu item under the Search menu.
2. Type the text ".+" in the Search for text box.
3. Type replace text " $0 " in the Replace with text box.
4. Click the regular expression checkbox.
5. Click the [Find] button to find the regex we need
6. Click the replace button.
7. Repeat steps 5 and 6 until we run out of regexes to search for.
8. Click the Close button to close the Search and Replace window.

3.2.2.1 Alternatives

Since we are trying to exercise all possible realistic ways a user might execute this task using this window, it's important to consider any of the reasonable alternatives available to doing the same task.

Alternatively to steps 4 and 5: One can

- Having clicked Find initially, replace and find successive matched expressions using the Replace & Find button repeatedly, and ending with clicking Replace to replace the final matched expression.

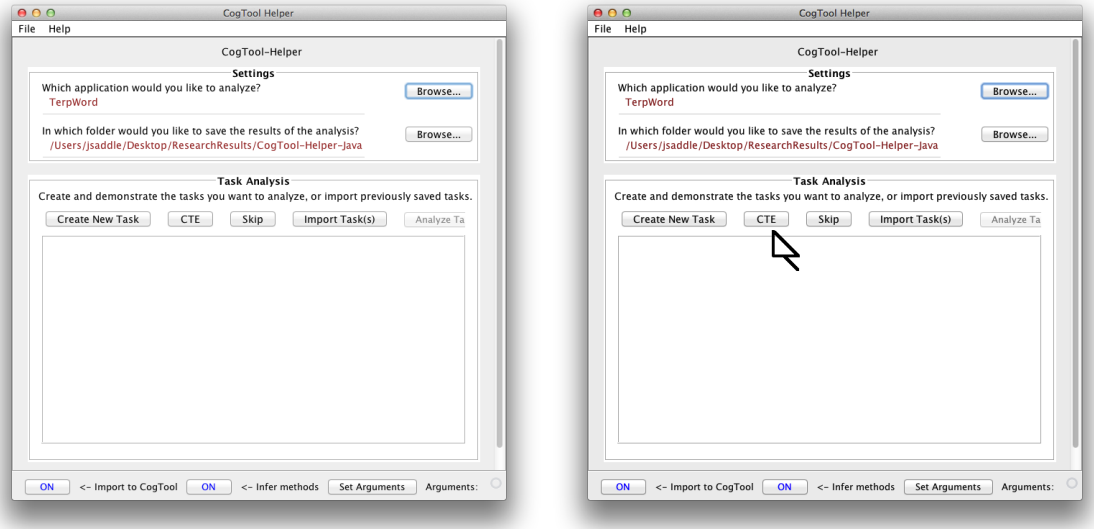


Figure 3.3: Start.

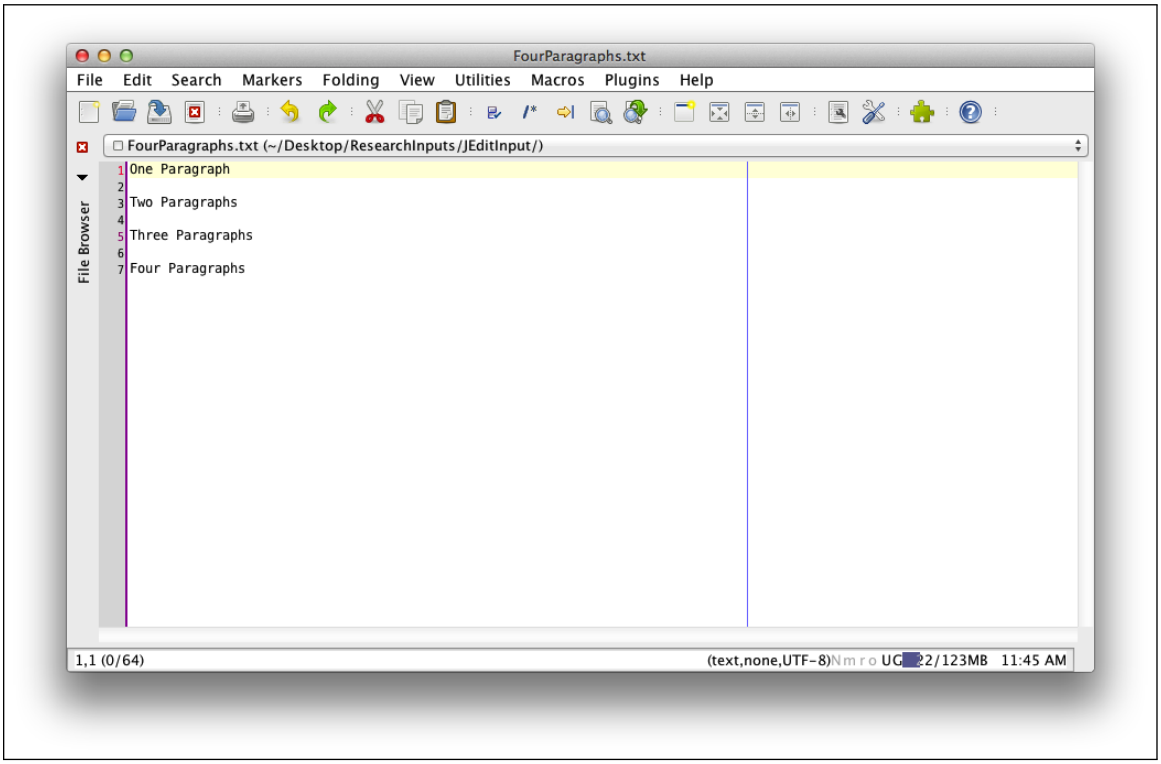


Figure 3.4: Initial Window

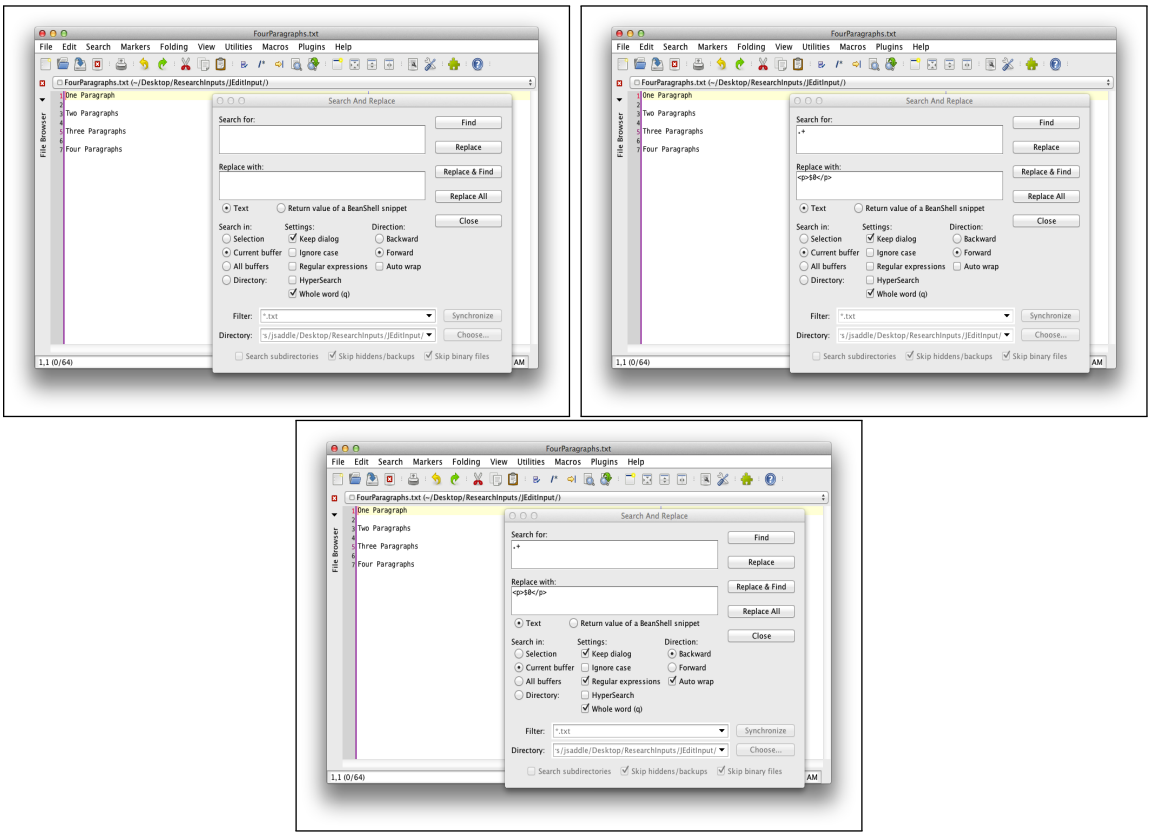


Figure 3.5: Find Window with various options configured: Nothing configured (left), find/replace text changed (right), Regular Expressions, Direction, and AutoWrap options changed (bottom).

- Change the direction at the start, select autowrap, before beginning either of the above options (step 3 can come before or after this step)

- Find and replace each regex using the replace all button once, eliminating steps 4 and 5 entirely from the primary option.

3.2.3 The Events-in-scope for Our Running Example

We will capture the following events using a point-and-click capture system that will be used to feed input to our generator. We will perform the following when the JEdit interface is opened in a point-and-click capture mechanism:

- Search menu (S_M)
- Search | Find menu item (SF_{MI})
- Regular Expressions checkbox (RE_C)
- Typing into Search for: text box ($SearchFor$)
- Typing into Replace with: text box ($ReplaceWith$)
- Find push button (F_B)
- Replace & Find push button (RF_B)
- Replace push button (R_B)
- Backward radio button (B_R)
- Auto wrap checkbox (AW_C)
- Replace All push button (RA_B)
- Close button (C_B)

We do not need to capture these events in any specific order while the point-and-click mechanism is recording our events. When we are finished capturing, the point and click capture mechanism will record what actions we've taken, and construct a file called the "events in scope" file that will hold our results until we are done with step C.

3.3 Step 2: Design a Logical Plan

After the tester can gather and visually lock in the names that the test case generator outputs to the user as they are captured, and after the tester has captured all events in scope, the next step is to design a logical plan that denotes constraints around some of these events in scope wherever they may appear together in any of the test cases in output.

3.3.1 Given Rules: Global Constraints

These rules are a "given" that govern every test case we generate. They are the same constraints used in HPRT [6]:

End in main window. The test case must complete with an action in the main window that does not exit the main window (Swearngin refers to this a "system interaction event") or with a terminal event that results in the main window remaining open.

Expand to Child. An event that expands a menu or a combo box must lead to the immediate selection of that parent element's child.

No repeat events. No event can reappear after appearing once in any test case (this constraint can potentially be overriden with a few of our user-defined constraints in special cases)

Window/tab open close will not happen. A window cannot be immediately opened and closed without performing any actions on that window. Also after expanding a tab, some other option other than the close button, must be selected within that window or expanded tab.

All these global rules protect against redundant and non-constructive behavior in test cases. In all our test cases, we want to focus on modeling the normal user who foremost interest in progressing through the task.

3.3.2 Introduction to Parameterized Rules: User-Specified Constraints

The logical plan points you can specify to our generator total to 5 different named constraint types that may be combined to generate a wide complexity of test suites. They are Exclusion, Order, Repeat, Required, and Atomic. The first four were used in the HPRT work [6]. The last constraint (Atomic) is new in EventFlowSlicer. From the user point view, we start with simple easy constraints, and move to the facets: Before looking at "harder" constraints such as the order events must appear in the generated test suite and how they must appear together, we'll begin with more obvious constraints such as specifying which events must appear in every test case at least once.

3.3.3 Required Constraint

Starting out in our logical plan is the required constraint. We define a single "Required" rule to specify that one out of a certain set of events must appear in each test case. As an example, consider that "Event A" and "Event B" were specified in the events-in-scope. The tester specifies with the required rule that one outcome for the test suite generated should be that for any given test case in the suite, either Event A or Event B should occur at least once in each test case. Each single "Required-type" constraint is specified as a set of events, and we may specify multiple of these required-type constraints to the generator if more than one type of event is required. To meet the tester's output specifications for the example here-stated, we would specify a required rule by creating a required rule

$$\{A, B\}_{REQ}$$

as input to the generator. If only event A must occur in every test case, we could instead specify:

$$\{A\}_{REQ}$$

3.3.3.1 Running Example

We're going to specify our logical plan as a sequence of constraints, C = C(EXC), C(ORD), C(REP), C(REQ), C(ATM).

(It should be briefly noted that the parameters are not shown in the logical rules we state in our logical plan for this reason: at the time of specifying the events in scope in the capture process, we bind the parameters we used to properly access widgets - such as locations of the buttons we click, and the text we type into the text box upon capture - to the events-in-scope and persist these along with the files we output. The parameters are thus bound and inferred with alongside events we captured. When the test case generator runs, it will then recognize these parameters and pair them with the events in each test case to ensure that the test cases we can replay contain valid parameters - as certain events in a test case such as text box events would not make sense without valid parameters, such as what text the tester typed.)

In our running example, we need to ensure that a subset of six key events of the ones stated above occur in each and every test case:

- The events of: Typing ".+" in the Search for: text box, and Typing " $0 " Replace with: text box, and Clicking on the Regular Expressions checkbox, need to appear in every single test case.

- One of either the Replace Push button, Replace & Find push button, or Replace All push button, must appear in every test case.

```
<Required>
    <Widget>
        <EventID>toggle button_img bold</EventID>
        <Name>bold</Name>
        <Type>toggle button</Type>
        <Window>TerpWord 3.0</Window>
        <Action>Click</Action>
    </Widget>
</Required>
<Required>
    <Widget>
        <EventID>menu item_Format|Align center</EventID>
        <Name>Align center</Name>
        <Type>menu item</Type>
        <Window>TerpWord 3.0</Window>
        <Action>Click</Action>
    </Widget>
    <Widget>
        <EventID>toggle button_img alignCtr</EventID>
        <Name>alignCtr</Name>
        <Type>toggle button</Type>
        <Window>TerpWord 3.0</Window>
        <Action>Click</Action>
    </Widget>
</Required>
```

Figure 3.6: Setting the Required Rule.

We can construct a logical plan containing the following rule to express this one constraint to our generator as a constraint meeting all the properties stated above.

$$C_{REQ} = \{\{SearchFor\}, \{ReplaceWith\}, \{RE_C\}, \{R_B, RF_B, RA_B\}\}$$

3.3.4 Exclusion Constraint

Next in our logical plan, we should focus on whether events in our test space should never share the same test case. We will now define formally the "Mutual Exclusion" constraint C_{EXC}. The Mutual Exclusion constraint is defined as a set of sets $X_1, X_2, ..., X_{n<\infty}$. Like the required constraint, each set X is a single rule containing elements that are events, and a second rule (a second set of events-in-scope that may not appear together) can be specified by simply adding a new set X to C_{REQ}. The

events in each set specified may not appear together in any outputted test case. The rule stating events C and D cannot appear together, is formally specified:

$$C_{MEX} = \{\{C, D\}\}$$

3.3.4.1 Running Example

In our running example, we need to ensure that Replace All push button doesn't appear alongside three other specific widgets in scope, the Find push button, the Replace & Find push button, and the Backward radio button.

Inferring these rules requires some domain knowledge about the functionality of JEdit: having typed text into the "Search for" and "Replace with:" text boxes first, clicking the Replace All button takes care of all subtasks of replacing each of the matched expressions in the document at once. Thus using the replace all button in any test case makes the use of the Find button, Replace & Find button, and the Backward radio button, no longer necessary.

To this end, the tester should add to the logical plan by specifying the constraint:

$$C_{MEX} = \{\{F_B, RA_B\}, \{RF_B, RA_B\}, \{B_R, RA_B\}\}$$

3.3.5 Partial Order Rule

Often when there are many events in scope, the tester might require events in each test case to have order and sequence. The tester should uncover all orders in which the events in the test suite may appear, and specify them in the plan.

We might want a subset of events within a test case to appear in a certain order, while others may either appear in the same order or not at all. We give this specification

```
<Exclusion>
    <Widget>
        <EventID>menu item_Format|Align center</EventID>
        <Name>Align center</Name>
        <Type>menu item</Type>
        <Window>TerpWord 3.0</Window>
        <Action>Click</Action>
    </Widget>
    <Widget>
        <EventID>toggle button img alignCtr</EventID>
        <Name>alignCtr</Name>
        <Type>toggle button</Type>
        <Window>TerpWord 3.0</Window>
        <Action>Click</Action>
    </Widget>
</Exclusion>
```

Figure 3.7: Setting the Exclusion Constraints.

the simple name of "Order" constraint. We will formally define the order constraint in terms of sequences of event sets. A single order rule O is an ordered sequence of sets each containing events. Each set in O is called an "ordering group" and contains events that belong in that spot in the sequence if all or some of the events defined in O ever appear together in the same test case. Given that the order rule properly defines a partial order on some of the events from the set of events-in-scope, events that come in latter sets of O must only occur after events that come in sets defined earlier in O.

Say we have events D, E, F, and J all in events-in-scope W. We want test case implied by the sequence (D, E, J) to be valid, but test cases such as (J, F), or (J, E),(J, F),or (E, F) should not be valid. The rule we could formally specify could be:

$$C_{ORD} = (\{D\}, \{E, F\}, \{J\})$$

Note that test case J, E is made invalid according to the constraint, because J comes after E in the order specified. Note that (assuming event A is also an event $\in W$, test cases (A, D, J), (A, F, J), (A), (J), and (A, E, J) are still valid. The

test case (J,D) is an invalid test case because it does not follow the order of widgets defined.

If we wish to define multiple orderings, in a manner similar to the previous rules, we may do so. All test cases must adhere to all the order rules defined, and those that don't are considered invalid and omitted from the output.

3.3.5.1 Running Example

For our running example, we suggest attacking the problem of executing find and replace properly with events in the correct order, by breaking some of the events in scope down into classes.

In any given test case, we must first ensure that the "Search and Replace" window is configured to search for the desired search text and configured to replace the expression found with the desired replace text. We say the following events all belong to the same order class - ergo they belong together - at the start of the test case: typing into the Search for text box, typing into the Replace with text box, clicking the regular expressions checkbox, and clicking the backward checkbox. Next before replacing any text, we must click the find button if it is necessary to do so. The Find push button comes in the next order class. At the end of the test case, we can begin to replace the text using the replace buttons, until we finish the task and are ready to close the window. The Replace & Find push button, and Replace All push button come in after the final order class.

We only need one order rule to specify all this, as follows:

$$C_{ORD} = (\{SearchFor, ReplaceWith, RE_C, B_R\}, \{F_B\}, \{RF_B, RA_B\})$$

3.3.6 Strict Sequences and Atomicity

The tester might require that if an event is to appear in any test case, that it only appear in certain appropriate places within a given sequence. Such a constraint might become the most useful of the five in filling holes other rules don't adequately fill for the purpose of generating test cases in your suite meeting very specific needs than the order rule alone can handle.

We will formally define this constraint, called the "Atomic" rule, as a sequence of event sets. Like order rules, we again use an N-tuple of the ordered components. Taking from our generalized example for order rules, we could reuse E, F, and J to create an atomic ordering (E, F, J), but this would mean that E, F, and J must occur atomically in that same sequence, in any outputted test case, with no substitutions or breaks in between. If any event element of this specification appears without the other two, and not in the order specified, the test case is invalid and omitted from output. Thus the sequence (E, F, J) must appear atomically if any one element is to appear at all. Multiple unique event elements appear in the same atomic-ordering set within the sequence as in order rules, however, only one of the events may be substituted for the one first specified, in that spot in the sequence.

In other words, say we have events S_1, S_2, S_3, S_4 that are "steps to a recipe". These four events must be carried out in order. Omitting one step from the sequence is not allowed.

Informally we can call the atomic rule a "recipe sequence" rule, where leaving out a member of the recipe, or performing a step out of order invalidates a test case.

The atomic rule is a sequence $Q = S_1, S_2, ..., S_{n<\infty}$, (n indicating the number of the final step in the sequence), where $S_1...S_n$ may consist of events from the set of

events-in-scope W. The rule specifies that each and every time an event of signature S_j from Q, appears in a test case, the following must hold for that test case:

- if $j > 1$, S_{j-1} from Q appears immediately before S_j in that same test case.
- if $j < n$, S_{j+1} from Q appears immediately after S_j in that same test case.

3.3.6.1 Running Example

To add atomic rules to our logical plan of how to attack find and replace, the tester needs some specific information about how the interface works. Note the alternatives to clicking Find and Replace:

- Click the find button, then replace button, then repeat the previous two steps until we run out of regexes to search for.

In our input file shown above, there are four paragraphs that need to be replaced. Amongst the few alternatives we have to get this to happen, a subsequence is involved in this alternatives requires the longest sequence of restricted activity among the three alternatives presented. The test case must follow the click of the Find and Replace push buttons, with three more Find-Replace back-and-forths before ending the test case.

- Having clicked Find initially, replace and find successive matched expressions using the Replace & Find button repeatedly, ending with clicking Replace to replace the final matched expression.

This alternative requires that we have test cases that follow the use of Find, then Replace & Find, with two successive clicks to Replace & Find, and a final click to Replace before ending the test case.

- Change the direction of the search and turn auto-wrap on before beginning either of the previous alternatives (any of steps 1-3 can come before or after this step)

This alternative requires that backward and autowrap accompany one another in any test case where either is found. Thus two alternatives are possible, backward and autowrap, or autowrap and backward.

(one could argue that backward and autowrap could potentially appear apart from one another, however we assume that the normal user might honor the correspondence between these two events, by clicking one right before the other. We argue that stranding one from the other doesn't make sense from a normal user standpoint in light of the fact that the test case cannot continue beyond a certain point without both.)

We can cover all these varied yet necessary alternatives with a set of three atomic rules:

$$C_{ATM} = (\{F_B\}, \{R_B\}, \{F_B\}, \{R_B\}, \{F_B\}, \{R_B\}, \{F_B\}, \{R_B\})$$
$$(\{F_B\}, \{RF_B\}, \{RF_B\}, \{RF_B\}, \{R_B\})$$
$$(\{AW_B\}, \{B_R\}),$$
$$(\{B_R\}, \{AW_C\})$$

Atomic rules are quite expressive. We have shown how atomic rules can be multiplexed by specifying two or more rules at once in C_{ATM}. There are special cases implemented in our generator for atomic rules however that don't apply elsewhere:

Two rules may contain the same event X. In this case, least one atomic rule

specifying X from the set of all atomic rules specified must be obeyed in that test case for every time X appears in a new atomic subsequence.

3.3.7 Unbounded Repetition

Recall that is a global rule in our test case generator that if any event appears more than once in any test case, the test case is invalid, as the test case exhibits redundant behavior. Thus, by default, each event is bounded to only appear once in every test case.

This rule can be overridden, however, in circumstances where the tester wishes to output a test cases were repeating one or a few elements within the test case is not redundant, but perhaps necessary to complete the task. We can specify a "Repeat" rule to meet this need. The content of a repeat rule is a single element, specifying which element of the events-in-scope may be repeated multiple times.

For instance, to click two separate menu items under a menu within the same test case, a the test case may need to open the menu twice to reach either menu item. To allow event $A \in W$ (say a menu situated above two required menu items) to repeat more than one time, we create the the repeat rule:

$$C_{REP} = \{A\}$$

3.3.7.1 Running Example

In our running example, we specify no repeat constraints.

However, note that some of our atomic rules repeat certain events in their specified sequences. Atomic rules can optionally override the No Repeat Global rule in a special way: Widgets that are not specified in repeat constraints can only be repeated as

```
<Repeat>
    <Widget>
        <EventID>menu_Format</EventID>
        <Name>Format</Name>
        <Type>menu</Type>
        <Window>TerpWord 3.0</Window>
        <Action>Click</Action>
    </Widget>
</Repeat>
```

Figure 3.8: Setting the Repeat Constraint.

their atomic rule specifies. Upon finishing that atomic sequence, those widgets cannot be repeated again. This allows us to repeat the find button 4 times, but not an unnecessary fifth time having completed the atomic sequence we specified.

3.3.8 "Omitted" Rules and Residual Effects

The reader may begin to notice that some widgets in scope have been left out of some of these rules for reasons not immediately obvious, such as the replace button in our plan's order rule, or the Auto wrap checkbox in the order rule.

However, the presence of the auto-wrap button actually is implied as being in the third class of the order rule (following the find button) even though we don't specify it. This is because in this example, we have a case of "implied rules". Our atomic constraint in our plan specifies that auto wrap must appear only when backward appears, immediately before it or after it. Notice that this necessarily implies that autowrap be in the same class as Backward in C_{ORD}. We don't need to be redundant in expanding the rule in the order constraint for class 1, because the constraint is already taken care of by the atomic rule. Our plan's Atomic rules also have an effect on the order relationship between the two push buttons Find and Replace.

Implied rules can have an impact on the amount of time spent specifying rules. It's good to point these trends out when devising the logical plan. It makes the step of

verifying the correctness of the test suite easier when there are fewer rules to consider as the culprit of problems.

3.4 Step 3: Generating the inputs and Running the Generator

In this step we devise and construct the "parameterized constraints", a plan derived from the logical plan constructed in step 2, that the generator can read in order to assist in the future steps of the process.

In our technique, we use a constraints-construction program (CCP) to prepare a "constraints file" from the events-in-scope output of the capture process and additional user input where the user supplies the user-specified constraints. The constraints file includes human-readable XML objects representing the events in scope, and the optional user-specified constraints that may group them together. The CCP generally makes it more easy to specify these constraints than hand-coding these XML files.

3.4.1 Generate Event Flow

The process of reading in constraints takes a few seconds. The next step is for the system to derive the model from the application we've specified. Model based testing literature calls this process: a "GUI rip". After providing the system with a few more arguments this time, such as the location of the Java application, the location of the jar files, the required classpath information, application specific arguments about what files to open during the rip, the system starts the process of mechanically opening the application and activating each widget it can find until it exhaustively searches the interface.

A version of the GUITAR ripper for JFC Swing applications we have implemented has been modified to take arguments that guide the ripper past certain widgets that can cause operating system or file system related problems, such as widgets that send documents to the printer, close the currently active application, or save random documents to the file system.

This process can take time depending on how deep and large the window structure of the application is. When the process is complete, our modified system prints out a copy of the successfully ripped GUI hierarchy file containing only widgets necessary to reach and activate the widgets we specified in our parameterized constraints plan. In addition, the ripper uses both pieces of information to generate an EFG (event flow) file that maps the allowed event flow between each of our widgets in scope. We'll need all these outputs later.

3.4.2 Verifying the Authenticity of Output

The tester will want to verify after this step that all the events that are connected in the outputted event flow graphs match the events that are intended to be present in generated test suites. For a sanity check, it would also be good to ensure that some of the flow (a path or so through the graph), expected to be present in outputted test cases, actually can be seen as present in the flow graph as a chain of directed edges.

As these flow graphs can be visualized using open source graph visualizers, this is easy and painless to verify before proceeding. If the graphs don't match what you would expect, now would be a good time for the tester to return to step A, and re-specify the events he actually wanted to see appear in the graph, depending on whether some events appearing in the visualization of flow appear inside or outside the scope of what test cases you plan to generate. As a rule of thumb however, there

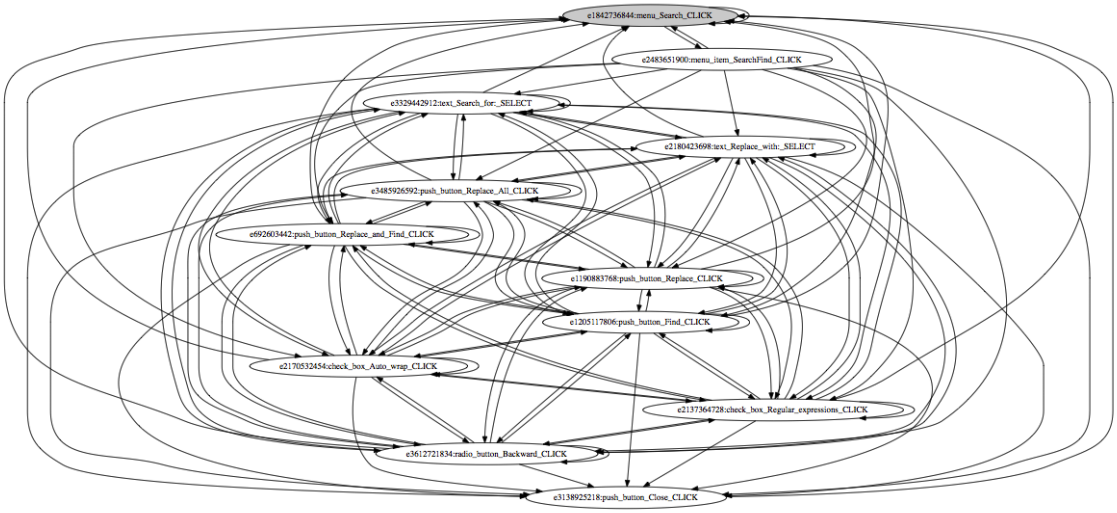


Figure 3.9: Our Generated EFG

are often events that may appear in the flow that the user might not expect to see, but that are required to reach the event in question. It's easier to see this when the flow is visualized.

The tester should then step B after reifying the events in scope, to regenerate the parameterized constraints for the system (the system needs all the events in scope to be present when generating parameterized constraints.)

The edges that appear between the nodes are what the tester does not need to worry about changing. The rules for generating these edges obey generally the algorithms implemented in GUITAR, which are specified in MemonGuitarRipping and NguyenGuitar.

Shown in figure 3.9 Here is a visualization of the EFG used in the running example. While EFG's typically come in a variety that doesn't give good labeling to go with the nodes shown in EFG visualizer's, we have instrumented a visualizer to show helpful names within event's respective nodes. EFG construction occurs immediately after the GUI rip, and typically takes a matter of under ten seconds to complete.

3.5 Step 4: Generate the Test Cases

It is at this point where the constraints are used to reduce the test case graph via global and user specified exceptional cases, and generate test cases based on global and user specified constraints. The system will take the parameterized constraints, and the event flow graph, and the tester will actually be able to see the output of the generator as it traverses a reduced portion of our test case graph to generate test cases the user specified via consraints. There are a few outcomes of this.

If the test case generator does not terminate, then the tester must stop the generator manually, and must go back and reify constraints in the logical plan. The constraints specified may have allowed to many instances of the "repeat" constraint, and perhaps mistakenly allowed a group of events in the graph to be repeated in a test case an unlimited amount of times in a never-ending cyclic fashion (the tester has to watch out for when a cyclic path exists in the visualized EFG from step D - the generator could end up in this described situation if it were to go down this or other cyclic paths). Having re-ified constraints, the tester can re-parameterize their logical plan, generate a new event flow with new nodes (if necessary), and finally re-run step E.

Since it isn't until this step where we actually use constraints from our plan to reduce the EFG's: the event flow graph generated previously may or may not change, depending on whether or not the tester went back and changed the set of events in scope. If not the tester does not need to re-run the GUI rip step C; time is thus saved in this case.

The test case generator may successfully terminate. If the generator terminates, the tester can be sure that the constraints specified were successful in giving the generator something that it could use to generate a closed-ended set of valid test cases, if any existed that the constraints would allow. The tester will want to then verify

that the test cases generated and stored to disk, validly represent what the tester would like to test.

There is text output that the generator prints to console, separated and in order of discovery via the DFS traversal procedure, along with names of the steps emulating the names within nodes from the overall event flow visualization that was optional in step C. The tools to verify test cases in graphical format are simple and powerful tools to use. They can visualize each test case individually as a "chain" flowing from start to finish, and can be used to combine all the test cases into a “pseudo-flow-graph” that shows how events flow from the first event in each test case, to the last event in each test case.

The nice thing is that this can be done for any suite you generate, yet the best choice between using visualizations and reading the text-based versions depends on personal preference, on the number of test cases you generate, and the variety of steps involved: in our validation of the test case generation method, text or graphical format did not always turn out to be a wiser choice than the other.

Having verified the test cases do not match the tester’s intentions, the tester would need to go back and reify the constraints, to see if they missed any events-in-scope, or missed any constraints.

However, having verified the test cases do match the tester’s intentions, the tester now has a full test suite. And coming soon, there will be a way to actually replay all those test cases, and test your program.

3.5.1 Our Running Example Output

Our running example produced a total of 114 test cases that represent how to use the find and replace window to search through and replace 4 instances of text in the JEdit text editor.

The process to create all this information is completely automated from the time the input construction program has compiled the rules file. The whole process following this point usually can take from 1 - 6 minutes to complete, depending on the complexity of the application being ripped and the size of other input parameters.

Chapter 4

Computing the Test Cases: A "Depth-First-Search"-Based Solution

A key idea that sets our test case generator apart from others is embedded in how we chose to improve on similar GUI test case generation algorithms. Inspired by modern implementations of fast algorithms, our goal was to implement an algorithm that traversed nodes in the EFG, while collecting information and parsing it in parallel, in a manner as fast as possible from start to finish. Our idea manifested in a depth-first search solution. It incorporates the use of all the consraints the user specifies as input, an uses a recursive descent strategy to deterministically decide upon test cases that match the user's specified logical plan for widgets in the EFG.

4.1 Introduction: The EventFlowSlicer "DFSGenerate" Algorithm

Swearngin in [6] modified the test case generator provided by the GUITAR GUI testing framework to allow it to support a set of user-specified output modifiers called

"constraints." Constraints are generator rules that allow the user to express logical bounds on the inclusion, exclusion, and moreover the order that events may appear in every test case, by grouping together constructs that each represent a widget both visible on the screen and relevant to the task while the task is occurring. In addition, Swearngin added global rules the test case generator would compulsively expressed as "a given", which constrain all test cases to behave in a special fashion that avoids redundant behaviour such as repeatedly clicking objects that can be toggled. While this last point relieves the user from the need to build mundane constraints repetitively, the ability to customize constraints gives the tester flexibility in controlling from the top-down both the behavior of test cases generated, and the amount to some extent. By allowing the user-tester to exercise different logical constraints amongst various widgets, the tester can experiment with and create different modelings for various situations where special behavior is needed, and let the flow remain unconstrainted among widgets where it isn't needed. Swearngin's HPRT generator was designed to capture the events and generate testcases only for software systems built upon the StarOffice/LibreOffice *UNO* Framework, which has its own distinct set of accessibility libraries for use by application developers of technologies based on LibreOffice.

In our work, we designed a new technique that takes the same inputs as the GUITAR generator [22] that utilizes rules similar to the generator proposed in [6]. Our goal was to support a wider variety of applications - in particular Java-based AWT/Swing applications. In addition, we wished to add rules that we believe make this generator even more robust in its ability to capture more scenarios that people wish to test that occur commonly in everyday applications, and to support the capture, replay, and review of test suites on applicaitons that are popular in the real world, using tasks that have real-world significance for these

4.1.1 Reduction Step

We want the graph to be as small as possible before generating test cases. In our implementation of EventFlowSlicer we brought together a set of pre-generation rules that pacify the input EFG graph "in-situ", just before running the generation algorithm. We call the implementation step, the "Graph Reduction step". Below is a shortlist detailing what the reductions are, based on a mix of global "given" constraints, and some information provided by the rules file passed in for the test case generation step that we can use before test case generation even begins:

IllegalSelfEdges. (code RS) Remove self edges of widgets not in the repeat set.

MenuToNonMenuChild. (code EC) Remove edges leading from an event specifying the opening of a menu to any event not a child of that menu.

FocusedWindowToTerminal. (code WO) Remove edges that lead directly from an open-a-window event to one that closes the same window.

It is shown in [10] that the complexity of an application has direct correspondence to the length of time it takes to generate test cases. We can also say that number of edges in an EFG directly corresponds to the amount of "application complexity". The amount of events in an EFG, the nodes in the graph model, make the potential size of that EFG larger due to the properties of the maximum edge set size of any mathematical graph being a function on nodes. The algorithm Memon designed, which builds an EFG based on the widgets available in a multi-window environment, implies in its core pseudocode, a crucial corrollary.

For a simple interface that has buttons, say one single window interface, and one with say one single button : suppose we want to add a single button to the one-windowed button. Each widget like that button we add to that interface gains at

least one edge leading from the old button to the new one, and from the new button to the old one, provided that neither button closes their shared window. What this means is that the number of edges is always growing with the number of nodes, and that additional widgets we add, if they share only a few windows amongst them, continuously grow the size of the edges, and if no windows are closed, grow the size of the edge set at an exponential rate.

Adding one button to the one-button window creates an edge set of size 2, but adding another creates one of size 4, another button creates 9 edges, the next 16, the next 32. It should become obvious to the reader after understanding this simple corrollary why EFG's become so complex after 9 or so buttons exist in a window's interface. Memon's method extends beyond single window interfaces, and thus the EFG construct simply becomes more complicated from there. Extensions we've made to GUITAR that build EFG's from the ground up retain this basic property of same-window edges between widgets.

Hypothesis: The EFG traversal must take at least $|V|^2$ (number of nodes squared) time to traverse and create test cases on widgets in a single windowed environment.

Base case: For a simple interface that has buttons, say one single window interface, and one with say one single button. Traversal of this graph to generate all test cases is 1 unit of time since 1 unit exists.

Induction step: We want to add an element to a graph with n nodes, an event, to the interface, and traverse the corresponding EFG for all test cases.

Induction hypothesis: An EFG traversal must take at least $|n+1|^2$ (number of nodes squared) time to traverse and create test cases on widgets in a single windowed environment.

Proof of Induction hypothesis: According to the EFG construction algorithm, for each unit event we add as a node to the EFG, we must connect a directed edge from

old nodes to the new node, and an edge going back from that node to that other edge. For normal buttons (that don't disappear when clicked), edges also lead from that button to itself. We already had n^2 edges in the graph. Thus we have n new edges going out of the new node, n edges entering the new node, and finally an egde leading from the node to itself. We've just added $2n + 1$ edges to the graph. But $n^2 + 2n + 1 = (n + 1)^2$. Thus we have just created a graph with $(n + 1)^2$ edges, and we have our desired result.

From this formal proof, it should be clear to see that it should take longer to generate test cases the more nodes (events) are detected in the reverse engineering of the interface. Since the number of edges increases, the number of potential test case paths for an exhaustive full search algorithm among these edges increases.

Thus the removal of edges should reduce the computation time of an unconstrained algorithm, because there are fewer paths to consider.

4.1.2 Explanation: Graph Traversal Technique

The DFGen technique aims to solve the problem: Locate all test case paths beginning at nodes in our EFG, call it G_{TC} that adhere to all given constraints and all user constraints.

The technique is a single pass algorithm that solves the problem stated above in process containing 4 phases: Initialize, Validate, Branch, and Print. The recursion of this *DFS* based algorithm occurs in the branch and validate phases.

The initialization and print stages are only concerned with preparing the variables for branch and validation, and preparing test cases for output to whatever endpoint the user has specified that test cases need to be sent to: the file system or as output to the terminal. These two steps are shown in Algorithm 2.

The algorithm finds each test case matching criteria as follows. First, leaving from some "initial" element of G_{TC} (marked with a boolean value in the input as an "initial node"), call it *node A* we find a single new node at an edge endpoint leaving the root, call it *node B*. We need data to pass to the first call to DFSVisitAndGenerate, where the other two phases are executed, so just after initialization, we call DFSVisitAndGenerate on these initial nodes along with the maps we created.

DFSVisitGenerate will then copy and use whatever information it got from the parameters. We will explain what these parameters are used for. $maps.eC$ is used to hold records about what, if any, exclusion parameters are currently active on the current path, indicated by p, that we're working with now. $maps.oC$ is used to handle information about what order parameters in use, and $maps.aC$ is used to handle information about what atomic sequence parameters are in use.

In the validation phase, phase one, we decide whether the movement from $A \rightarrow B$, the last node in the path to the current one, i_new, explicitly violates a constraint, and if it does, we return from the call to visit that node B, and move to either that child's next sibling B_2, or to visiting a different node according to depth-first order in the EFG, and *repeat* the validation step. If we are currently visiting an initial node in the EFG, then we move on to another initial node in the EFG, until there are no more initial nodes to generate test cases from. That is: if the movement from $A \rightarrow B$ followsall exclusion rules AXB, Order rules, $k_A > k_B$, Atomic Rules, $AprecedesB$, and Repeat rules, $A = B \rightarrow A > A$, we continue on to the second phase, within the same call to DFSVisitAndGenerate. Passing these initial tests guarantees that there is no reason why continuing from p to reach b from p must generate only invalid test cases.

In phase two (Branch), we check a few more requirements to discover whether the path formed by adding i_new to *pathp*, and immediately branch off from the parent

Algorithm: DFSGenerate
Input: G_{TC}
Output: List(TestCase)
Let list of test case paths $allTests$ be initialized to $[.]$;
Let integer e become |ExclusionSets|;
Let integer o become |OrderSets|;
Let integer a become |AtomicSequences|;
Let eC, oC, aC each be maps initialized with e, o, a keys respectively, (one for each key in their respective set), each mapped to an initial value -1. Store all maps in a unit collection of maps, call it *maps*. *Each can be expressed as an array over e cells, each cell mapping to an integer -1 or [1..e].* In addition there is a final map in *maps* called aC_active, initialized with a keys, each key mapping to an enumerated constant, which can take on any value of the following:

$$\{NOT\_DEAD, DEAD, FINISHED, DEAD\_FINISHED\}$$

Each value in ac_active is initialized to NOT_DEAD

begin
  for $k \leftarrow 1 \ldots |G.V|$ **do**
    if *k is an initial node in* G_{TC} **then**
      $allTests.add$(DFSVisitAndGenerate(V[k], $maps$, $[.]$));
    end
  end
  for $k \leftarrow 1 \ldots |allTests|$ **do**
    $printToFileSystem(testCases_k)$
  end
end

Algorithm 1: Top-Level Operations: Initialization and Print

DFSVisitAndGenerate
Input: Vertex i
Input: Map Collection *maps*
Input: List of Integers p
Output: List(TestCase)
• **First:** Copy all variables from maps into a fresh map variable in local-scope: $maps\_new$.
• Copy list p into new list variable p_new.
begin
  ≫ **Phase One: Validate**
  for *each GroupType in the set* $\{EXC, ORD, ATM\}$ **do**
    if $testAndMap(GroupType, i\_new, maps) = false$ **then**
      return empty list $[.]$;
  end
  ≫ *Check Repeat*
  if $CheckForTooManyRepeats(i\_new, p\_new)$ **then**
    return empty list $[.]$
  end
  ≫ **Phase Two: Branch**;
  p_new.add(i_new);
  Let *testCases* be an empty locally-scoped list of test case paths
  if $EndsInMainWindow(p\_new)$ ***and*** $CoversAllRequiredEvents(p\_new)$ ***and*** $DoesNotOpenAndImmedCloseWindow(p\_new)$ ***and*** $FulfillsAtomicRequirements(p\_new)$ **then**
    testCases.add(p_new)
  end
  for *Vertex* $j \in outgoing$ **do**
    testCases.appendAll(
    DFSVisitAndGenerate(j, maps_new, p_new))
  end
  return testCases
end

Algorithm 2: DFSVisitAndGenerate function.

to find more test cases along paths that include the *pathp*, or the ancestry we just descended. We check to see if the atomic sequences are complete - if any were started along that path - and that all of the global rules other than repeat are met: that the path ends in the main window of the application, that the path meets the required rules rules, that the path does not open and close a window or tab.

If we can answer yes to all these questions, we add the path traversed as a test case to the output test suite. If no, we do not add the test case to the current path.

In either case, passing the first phase guarantees us access to branching out from the current node B, to its children, and running phase one again for each child of the current node.

At each call to DFSVisitAndGenerate, a *copy* of the parameters is set aside to run tests on, and to then pass down to the new copy of DFSVisitAndGenerate that will run the base of the second phase. This is done specifically due to a special invariant that shall hold at the beginning of each call to DFSVisitAndGenerate: that each child of a valid depth first path representing a test case, will be considered for inclusion in a test case formed by appending that child to the path — based on constraints-related information about itself and its ancestry, yet not its siblings.

To accomplish this, we hold active constraint data in maps we pass in the *maps* parameter passed to the call. We hold data about ancestry in the list of integers, p. We can simply create a copy of each map, store it locally within each activation record **and before calling DFSVisitAndGenerate on child** $node(B$, and pass this copy of the maps down to children. This copy we use for the children is thus held out of reach from being modified by children of $nodeB$. We can reuse data about ancestry of B, when calling DFSVisitAndGenerate on siblings of B. We can create new data about test cases containing B in their ancestry, and pass it down to children of B. to This is a source of memory savings for the implementation of the algorithm.

Provided that all guarantees hold, this method efficiently uses *DFS*, proven to exhaustively search a graph to find downstream nodes, to find all nodes that it can find leaving initial nodes in G_{TC}. It furthermore correctly records paths leaving those nodes that represent valid test cases via its second phase. If we end up failing the first phase in any call to DFSVisitAndGenerate, we are in effect pruning all path sub-trees lying downstream of the current node. This adds up to be a substantial time savings for the implementation of this algorithm.

However, there is a downside to being able to find all test cases adhering to constraints without specifying a length. Due to the fact the tester is allowed freedom to specify which nodes may be repeated indefinitely in a test case or not, the algorithm may fail to terminate if a special condition holds where two nodes that may be repeated indefinitely are joined by any cyclic path that obeys all other constraints.

4.1.3 Key Differences between EventFlowSlicer-Algorithm Generator and HPRT-GUITAR Generator

Length Parameter Tradeoff The HPRT generator takes different parameters than EFS. Using HPRT generator ensures that we enumerate all test cases that adhere to constraints a given length from any root node in the EFG. However, since the technique enumerates *all possible* test cases of a given length from a root, we are essentially discovering all paths leading from root to leaf at depth n from root, and repeating this process for however many roots there are in the EFG. This has the effect of blowing up the running time of this algorithm when exploring spaces of all possible ways to reach nodes in that graph using paths of length n, whenever n is very large. This blow up can be seen reflected in our running times chart.

EventFlowSlicer takes all the parameters HPRT generator would take, minus the

length parameter. We enumerate all test cases of many lengths at once, without the need to run the program more than one time to get all test cases, and without the need to know the length of each test case in advance.

However, since the generator is designed to enumerate all possible test cases obeying constraints without that hard stopping criteria, it will not halt in the case of traversing a cyclic path downstream from an initial node, in which all paths along that cyclic path (or an infinite number of them), are also valid test cases according to constraints specified. This has a sure effect of causing the call stack to overflow eventually, and in the case of our implementation of the algorithm based on Java, no test cases will be outputted to the file system, since the print phase takes place just prior to ending a valid call to the generator.

More Required Constraint Freedoms Both the HPRT generator and EFS generator take as input Required constraints, but the parameters allowed to be passed into the different generators differ. The HPRT generator allows the tester to specify that one or many elements must be required to appear in every test case. The DF generator allows for this to be expressed, in addition to the ability to express that each of the one or many elements required, may have up to *two or more distinct alternatives* for each test case.

Addition of the Atomic Constraint The EFS generator accepts atomic constraints in addition to the constraints supported by HPRT generator, of the form $A1 = \{S1, S2, S3, S4, ...Sn\}, A2 = \{...\}$ where each S contains a set of alternative events that can appear in the sequence at the nth spot in that sequence A. The only limitation in our implementation is that we do not support any instance where S1 of any defined atomic sequence overlaps S_N of another sequence (where any two atomic sequences defined as constraints form an overlapping chain).

```
TestAndMap
Input: GroupType scenario
Input: Vertex i_new
Input: Map collection maps
Output: Boolean value true/false
begin
    if scenario = EXC then
        for Group k ∈ excludeGroupsOf(i) do
            if maps.eC[k] ≠ −1 and
             maps.eC[k] ≠ i_new then
                return false;
            else
                eC[k] ← i_new;
                return true;
            end
    if scenario = ORD then
        for GroupSet k ∈ orderSetsOf(i_new) do
            let Group groupOfINew ← k.groupOf(i_new)
            if maps.oC[k] ≠ −1 and
            oC[k] > groupOfINew then
                return false;
            else
                eC ← groupOfINew;
                return false;
            end
    Continued in Part 2
```

Procedure TestAndMap: Part 1.

```
begin
    Continued from part 1
    if scenario = ATOMIC then
        let booleansomeGood ← false
        let booleansomeBad ← false
        for SequenceSet k ∈ atomicSets do
            if aC_active = DEAD, DEAD_REPEATING then
                someBad = true;
            end
            if aC[k] = −1 then
                if i_new ∉ some Atomic Sequence then
                    if isEmptyMap(aC) then someBad ← true;
                    else someGood ← true;
                    continue loop
                end
                if i_new ∉ AtomicSequence_k then
                    someBad ← true;
                    continue loop;
                end
            end
            • let collection of integers iGroups
              ← GroupA ∈ AtomicSequences_k wherei_new ∈ A
            if GroupaC[k] + 1 ∈ iGroups then
                someBad ← true
                aC[k] ← −1
                if aC_active[k] = NDEAD then
                    aC_active[k] ← DEAD
                else if aC_active[k] = FINISHED then
                    ac_active[k] ← DFINISHED
                end
                continue loop.
            end
            aC[k] ← aC[k] + 1
            someGood ← true
        end
        return (!someGood and someBad)
    end
end
```

Procedure TestAndMap, Part 2.

Chapter 5

Feasibility Study

In this next set of discussions, we will examine three tasks, explaining what the target test suite should consist of, and defend using combinatorial mathematics how each test case should look.

For each goal we present, our test case generator created a certain amount of test cases that each meet the goal. In addition to re-generating all goals specified in the HPRT genertor paper [6], we generated a total of 498. Having provided snapshots for the reader to understand specifics about the interfaces we're dealing with, we will defend why we believe the number of the test cases we retrieved from the generator is correct in completely representing all feasible combinations of individual steps we described earlier to achieve our goal.

5.1 JEdit "Four Paragraphs"

The task depicted in Figure 5.1 depicts the entry of search and replace parameters using text boxes and other selection options visible in the "Search And Replace" window to ensure that the paragraph text we're looking for can be found. Following,

JEdit	In a pre-created .txt file containing four separated lines of content: Open the "Search And Replace" Window
(FourParagraphs)	In the Search and Replace Window: Enter the regex ".+" in the "Search for:" text box, and the replace string " $0 " in the "Replace with" text box.
	Find and replace the four lines in the document with new text wrapped in tags, using the push buttons and options present in the Search and Replace window, or the JEdit menu bar.

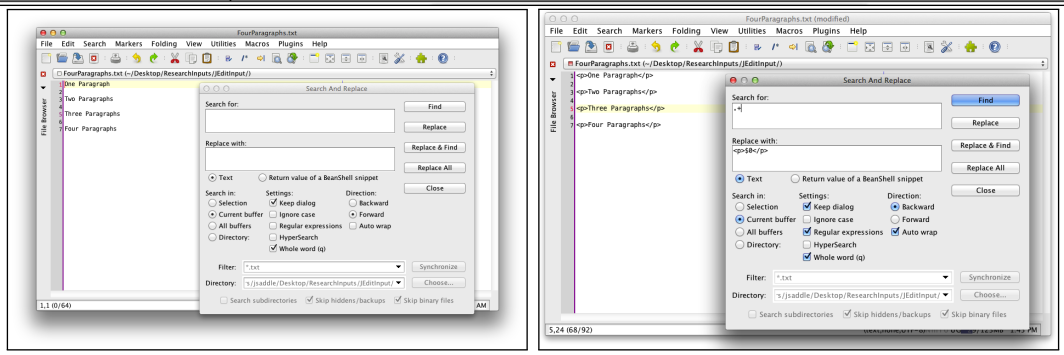


Figure 5.1: Test Description of the Task "JEdit Four Paragraphs". Snapshots of the Task.

the user must proceed using document selection buttons to select and replace each instance of paragraph encountered in the input file. All these widgets are accessible from the "Search And Replace" window shown in Figure 5.1.

The task involves wrapping some text in formatting markup used to mark paragraphs of text. Four buttons to the right of the text boxes are directly related to the function of changing properties of the input file defining this task's completion. Given the right order of clicking these buttons, they change the text incrementally or all at once from an initial state with no text marked-up, to a final state where all four non-empty lines contain markup.

This task is complex in that it involves numerous alternative routes of user input through this menu to achieve the goal. What makes this particular task special is that it involves a multi-step process and involves a variety of types of widgets in each version of the task. Performing this tasks modifies the beginning, middle, and end of an entire text document. We predicted and found that, if done correctly, all 114 ways

discovered by our test case generator should convert the document text from that of the initial state, to the document text of text declared to be in our final state, each and every time.

Finally, given the plethora of buttons in this particular window alone that aren't used, this task exemplifies our need to limit widgets-in-scope to those absolutely necessary to complete our one task, in order to constrain the number of test cases we must review later for correctness. Widgets left out of our selection were left out intentionally, though they can potentially be included and omitted without affecting the final state (such as [Ignore Case] checkbox). As a reminder, our intention is to discover efficient ways to accomplish the task. Our test cases cannot be bothered with widgets that do not lead to quick and efficient modification of the output, without risking unnecessary blow up of output. [Ignore Case] offers little to no functional advantage to any of our test cases, though it can be added in anywhere. We wish to minimize the effort the user must exert in reaching the end goal state whereever we can.

We're going to denote the text box underneath the label entitled "Search for:" using the symbol [Search for:] and likewise for that box under the label Replace with (using the symbol [Replace with:]). We're going to denote the regular expression checkbox [REX], the Backward radio button as [B], the Auto Wrap checkbox as [W], and the close button as [Close]. The document function buttons to the right of the text boxes, that perform changes on the open document, will be specified using their labels [Find] [Replace] [Replace & Find], and [Replace All]. Here are some variations on this task.

The user must utilize [Search for:],[Replace with:], and [REX] before utilizing the document function buttons to manipulate the open document. The user can swap the ordering of entering text into [Search for:], entering text into [Replace with:], and clicking [REX].

Length 9 test case (_I)	Length 9 test case (_V)	Length 9 Test Case (AM)
0: e1842736844:menu Search CLICK	0: e1842736844:menu Search CLICK	0: e1842736844:menu Search CLICK
1: e2483651900:menu item Search|Find... CLICK	1: e2483651900:menu item Search|Find... CLICK	1: e2483651900:menu item Search|Find... CLICK
2: e3329442912:text Search for: SELECT	2: e3329442912:text Search for: SELECT	2: e2180423698:text Replace with: SELECT
3: e3329442912:text Search for: SELECT[TextInsert .+]	3: e3329442912:text Search for: SELECT[TextInsert .+]	3: e2180423698:text Replace with: SELECT[TextInsert $0]
4: e2180423698:text Replace with: SELECT	4: e2137364728:check box Regular expressions CLICK	4: e3329442912:text Search for: SELECT
5: e2180423698:text Replace with: SELECT[TextInsert $0]	5: e2180423698:text Replace with: SELECT	5: e3329442912:text Search for: SELECT[TextInsert .+]
6: e2137364728:check box Regular expressions CLICK	6: e2180423698:text Replace with: SELECT[TextInsert $0]	6: e2137364728:check box Regular expressions CLICK
7: e3485926592:push button Replace All CLICK	7: e3485926592:push button Replace All CLICK	7: e3485926592:push button Replace All CLICK
8: e3138925218:push button Close CLICK	8: e3138925218:push button Close CLICK	8: e3138925218:push button Close CLICK
--	--	--

Figure 5.2: Sample Text Cases from the task "JEdit Four Paragraphs"

The user may utilize one of the following sequences to replace the four paragraphs with the proper replace text.

- Click [Find], click [Replace], click [Find], click [Replace], click [Find], click [Replace], click [Find], click [Replace]
- Click [Find], click [Replace & Find], click [Replace & Find], click [Replace & Find], click [Replace]
- Click [Replace All]

The user may opt to reverse the order that JEdit finds and replaces text in the first two alternatives above, by selecting ***both*** *the [B] and [W] options*

Finally we specify that each test case will complete by closing the "Search And Replace" window, using the [Close] button below the document function buttons.

We provide just a few test cases below as an example: Each test case clicks the [Search|Find...] menu item in the main window menubar, opening the "Search And Replace" before moving on to the meat of the task. Immediately the test cases adjust the parameters of the tasks using the [Search for:] [Replace with:] text boxes, and the [REX] checkbox, and then ends by clicking [Replace All] and closing the window.

In our example, it is shown clearly three of the total ways how the orderings of [Search for:], [Replace with:], and [REX] can be made concrete (two events represent

each of either [Search for:] or [Replace with:] - representing the fact a text box must be both selected then typed to in an atomic fashion). They are all valid orderings of these three widgets, used to set up the find task.

We'll note a few constraints we used in this task. We stipulate in the constraints an *order rule* so that these three widgets will come before any invocation of the [Find] or [Replace All] buttons. All three widgets are required to occur before searching for text within the document, and so 3 added *required* rules ensure that the three widgets occur first in each and every test case.

5.1.1 Defense of Test Suite Correctness

We do not specify a constraint for the order in which these events occur, as every ordering of these three is fair. There are in total 8 ways to order [Search for:] [Replace with:] and [REX], (in other words, ${}_3P_1$ or 3!). If we opt to continue the task by replacing each segment of text one by one in a "forward search", moving forward through the document to replace text, there are then 3 replacement strategies that can follow each of the 8 setup strategies. If we were building our formula from the bottom, we would then have the incomplete equation,

$$3! * 3$$

for our current set of calculations.

Now recall that there exists mid-window a radio-button-checkbox pair ([B] and [W]) that can cause the search to find and replace segments of text, to go in the "reverse" dircection. This two button combo may be used at any time during task setup. So we now have 24 (4!) ways in which to order all three of our setup widgets plus our new pair - but since both [B][W] and [W][B] are both, we must account for

all the ways that [B][W] or [W][B] might happen, and so we have 48 total ways (4! * 2) to reverse the direction during setup before jumping into a replacement strategy.

Our formula becomes:

$$3! * 3 + 4! * 2 * (?)$$

Finallly note that the function of replacing all the text at once is omnidirectional (the selection for replacement can really be said to move neither forwards nor backwards, because the button replaces all matching text at once). Test cases containing occurrences of [W][B] or [B][W] alongside [Replace All] are considered invalid from our standpoint, as [W][B] does nothing to functionally change the output of Replace All, and thus is an inefficient use of the reversal feature) and are thus removed from the output test suite using constraints. Instead of three replacement strategies for the backward cases, we now have two. So our formula becomes.

$$3! * 3 + 4! * 2 * 2 \text{ ways...}$$

$$= 6 * 3 + 24 * 2 * 2$$

$$= 18 + 96$$

$$= 114 \text{ ways.}$$

And we have the final result justified.

Shown in Figure 5.3 is an EFG depicting the minimal test case graph, showing the progression of all test cases from the first Search Menu event to the final replace option chosen, down to the close button at the bottom.

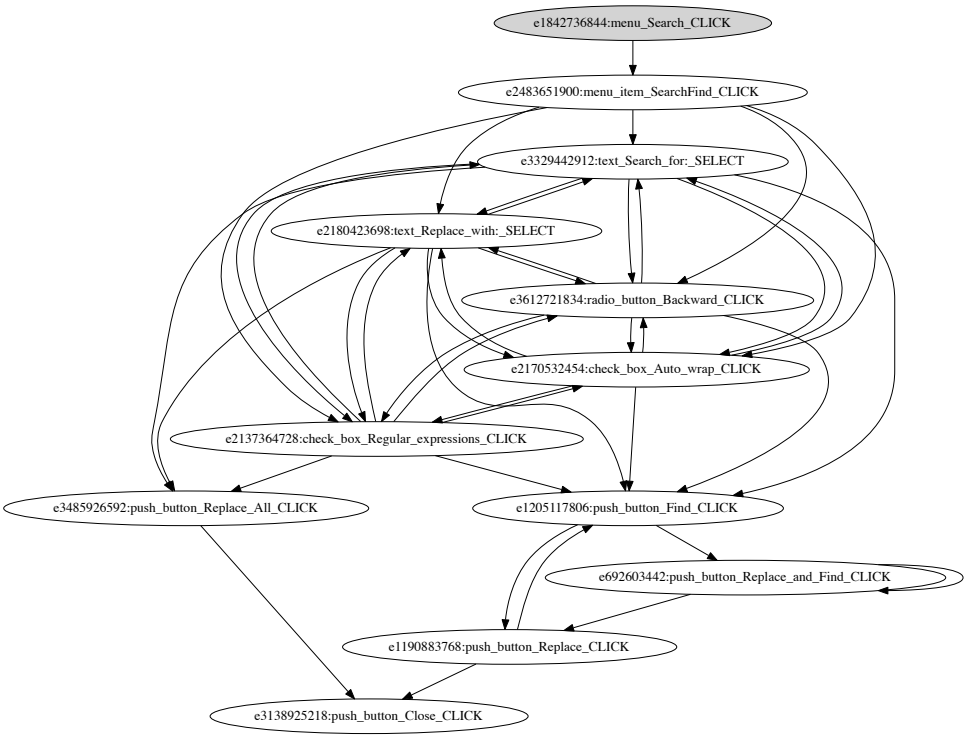


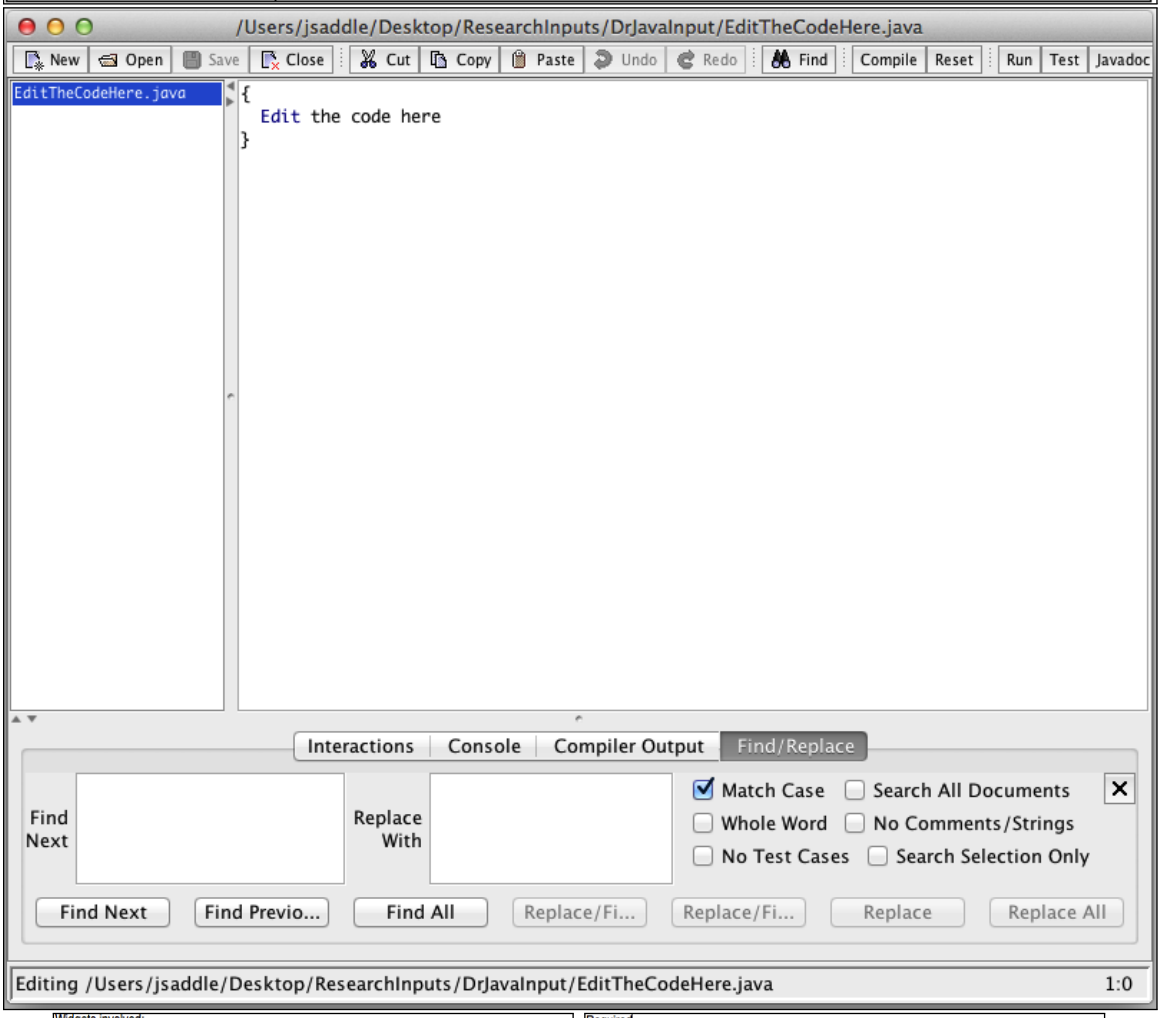
Figure 5.3: Test Case Graph with edges taken from test cases of Four Paragraphs

5.2 DrJava "Search Options"

This next discussion focuses on why we chose to generate the test suite specified in our task "Search Options", executable on the Dr. Java IDE. This task's consraints specify that certain options are to be used to demonstrate the removal of some text, and its replacement by new text we provide as parameters to Dr. Java's search-find-replace functionality.

Our goal by defining the task using the events in scope as shown above, was intentional. In the planning stages, we really wanted to seize the opportunity to model a complex task in Dr. Java's interface. A search and replace task with many options, using DrJava's search-find-replace functionality seemed like a readily availble opportunity to do so at the time. On the other hand, we admit there are about 7 more

DrJava	In an open document containing the text "Edit the code here": Open the Find/Replace Panel in the "Interactions Pane."
(Search Options)	In the Find/Replace Panel: Enter "Edit" in the "Find Next" text box, and "Replace" in the "Replace With" text box.
	Search for and highlight a single word specified in "Find Next".
	Replace that text with the text typed in "Replace With".



Widgets involved:
1. Find Toolbar Button
2. Edit Menu
3. Edit | Find/Replace Menu Item
4. Match Case Checkbox
5. Whole Word Checkbox
6. No Comments Strings Checkbox
7. Find Next Button
8. Replace Button
9. Typing "Edit" into Find Next Text Box
10. Typing "Replace" into Replace With Text Box
11. Interactions Page Tab List Button (option 3)

Total 11 Widgets

Required
Rule 1: Edit | Find/Replace Menu Item ∨ Find Toolbar Button
Rule 2: Typing "Edit" into Find Next Text Box
Rule 3: Typing "Replace" into Replace With Text Box

Required
Rule 1: Edit | Find/Replace Menu Item ∨ Find Toolbar Button
Rule 2: Typing "Edit" into Find Next Text Box
Rule 3: Typing "Replace" into Replace With Text Box
Rule 4: Find Next Push Button
Rule 5: Replace Push Button
Exclusion:
Rule 1: Edit | Find/Replace Menu Item X Find Toolbar Button
Order
Rule 1:
G0: Edit | Find/Replace Menu Item ∧ Find Toolbar Button
–> G1: Find Next (Nameless 5) ∧ Replace With Text Box (Nameless 6)
–> G2: No Comment Strings Checkbox ∧ Whole Word Checkbox ∧ Match Case Checkbox
–> G3: Find Next Push Button
–> G4: Replace Push Button

Repeat: (none)
Atomic: (none)

Total 7 Rules

Figure 5.4: Test Description of the Task "DrJava Search Options". Snapshots of the Task.

widgets available depicted in Figure 5.4 that could be used to perhaps accomplish the same task as the 7 we chose to include in scope for this task. The number of test cases including all 14 would certainly get out of hand.

Search options still uses a find-and-replace search *window*, similar to JEdit. Unlike JEdit however, Dr. Java's search window opens up in the window opened during startup. The bottom half of the window transforms to show a both a new tab and panel, the "Find/Replace panel", which accompanies the main editor within the editor's window. With this task, we wish to exemplify how our technique can generalize to accommodate applications with windows that transform to show new content over time. We believe that transform in Java Swing applications is a common thing, and so design decisions for the replayer considered this. To appeal to this argument, recall applications that have configurable options menus, tabbed panels that can be hidden and shown, and moreover toolbar buttons whose positions can be configured at runtime. JEdit, TerpWord, Dr. Java for example, all have tabbed panes; DrJava and TerpWord both contained some that are key and essential to the tasks we've selected for these applications. We've built functionality into our replayer implementation to support this mighty trend among Swing applications.

We decited to constrain the space of possibilities in which this task could be completed for reasons of brevity, our primary goal being to attempt to showcase our test case generator's ability to handle transforming interfaces. Even so, we found that our constraints still led to the the generation of 64 test cases, and at the design stage, we believed this to be plenty enough of a convincing argument for this being a complex enough task. The options we did choose to include we found would collectively change the outcome of the test case depending on the way each option independently works, and in the way they work co-dependently. The requirements for finding the word "Edit" in the input document file are met by any combination of the three checkbox inputs

we use for this task: our search matches the word “Edit” in the input file because a) “Edit” is a whole word found in the input file, b) we entered “Edit” in the “Find Next” box with an alphabetic case matching the case of the term in the input file, and c) the term “Edit” does not appear in a Java comment or Java string literal.

Though there are many possibilities for searching we generated, “Search Options” has a simpler model than that of “JEdit Four Paragraphs” for generating test cases, as reflected in the constraints we built for this task. We will explain the alternatives within this test suite, and how we model all possible ways this test suite can be carried out using the widgets selected. We’ll refer to the Find Next button at the left side of the Find/Replace panel menu using the symbol [FN], the Replace button toward the right with the symbol [R], the Match Case checkbox with the symbol [Ca], the Whole Word checkbox with the symbol [Wo], and the No Comments/Strings checkbox as [Co]. For the search parameter text box adjacent to the label “Find Next:”, we’ll use the symbol, [Find Next:], and for that adjacent to the label “Replace With:”, we’ll use the symbol [Replace With:]. To show the “Find/Replace” panel, the user can either use the “Find/Replace...” menu option under the Edit menu, [EFM], or the Find button on the application’s toolbar [EFB]. Here are the variations of the task:

We may choose to show the "Find/Replace" Panel using either [EFM] or [EFB]

We may fill out the [Find next:] text box first, then [Replace with:], or fill out [Replace with:] followed by [Find Next:]. We always ensure this step precedes the selection of any future search-parameter-modifier checkbox, or the [FN] and [R] buttons.

Before searching for text using [FN], we optionally may click either [Ca], [Co], or [Wo], or a combination of any of the three to modify search parameters.

Each test case must end with clicking [FN] then clicking [R] to replace the search text in the main window. All alternatives considered, we are led to consider a total of 64 test cases in our test suite.

It might not seem obvious why there are 64 test cases. Let's take a closer look.

We'll break the problem down into parts. The first part we will consider is the fact that we may begin the task by either clicking the "Find/ReplaceâĂę" menu option or the Find toolbar button to open the panel and begin setting search parameters. Our formula thus starts out:

$$2.$$

Now we need to either enter "Search parameters" followed by "Replacement text" or vice versa, giving us two ways to follow either prior option we've just selected. Our formula is now:

$$2 * 2.$$

The final part of calculating our final test case count considers the variations of the search options available before clicking the find button. There are 3 in total [Ca] [Co] [Wo]. We may also click none of these options, proceeding directly to the Find Next button - and we'll call this option, option [e]. Our first choice consists of picking from these first four choices.

We'll follow this thread of thought to calculate how many choices we have remaining. Upon picking option [e] and clicking [FN] we have no more choices âĂŞ we must immediately progress to the final call to [R]. The second choice after picking any of the other 3 options, is any 2nd button among the two left over that hasn't been clicked yet, and of course utilizing option [e] to just move on to the [FN] button. This makes up for 4 concrete second choices after 4 concrete initial choices.

Following that second choice that does not progress directly to the Find Next

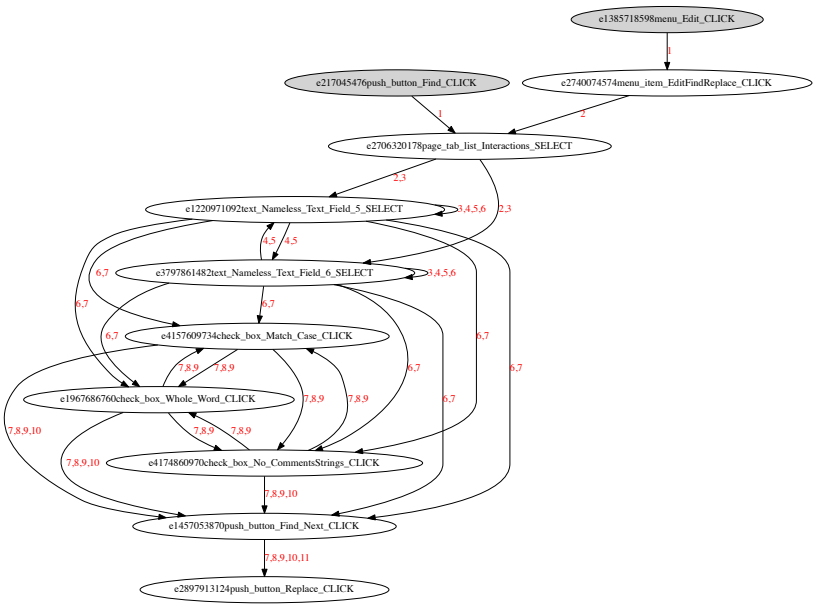


Figure 5.5: Dr Java Search Options Test Case Graph

button, we must pick the only option left over, and then we must proceed to click the find next button to finish the task. Our formula thus becomes

$$2 * 2 * (4 * 4) \text{ ways.}$$

$$= 4 * (16)$$

$$= 64 \text{ ways.}$$

And we have the final result justified.

In the special diagram depicted in Figure 5.5, we number each step as we progress through each DrJava test case (each step from event to event is nnumbered only once), to create a flow chart resulting from the combination of all the test cases into one concrete graph. If we label each edge with a number we can clearly see that the user starts with a certain step, then continues to another then another.

JEdit	In a pre-created .java file containing a some text formatted as a line comment: Open the "Style Editor" Menu
(CommentedText)	Change the appearance of all comment text in the open JEdit document, using the options made present in the style editor menu. Confirm the changes.

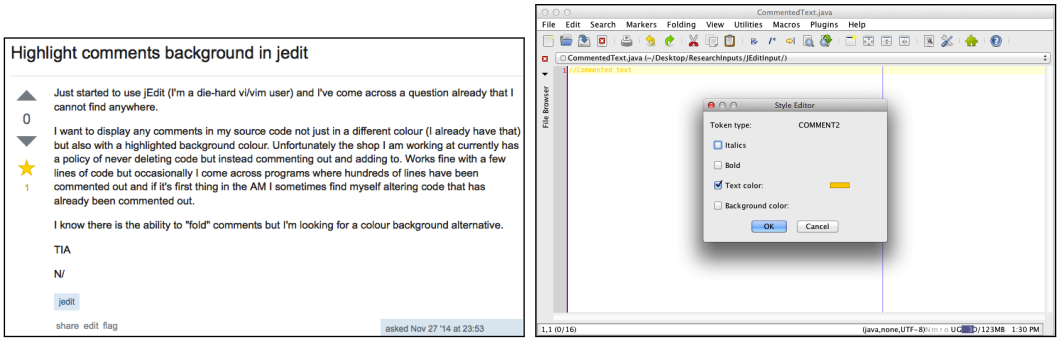


Figure 5.6: Test Description of the Task "JEdit CommentedText". Snapshots of the Task.

5.3 JEdit "Commented Text"

This next discussion concerns the reasons behind why the test suite JEdit "Commented-Text" became a good candidate to use for our experiments. Some elements of the task this test suite is attempting to model are directly driven by the purpose of answering a question taken from on StackOverflow during our motivating research phase, regarding how to change the background color of all text meeting certain criteria within the JEdit application. In this case, the original poster of the question had chosen a readily available attribute assigned to certain text within a Java source code is opened within the editor. By default, in the installation of JEdit, text that is preceded by a Java Line Comment Marker (//) is colored orange and is given a transparent background color. The original poster's question regarded changing this default behavior of JEdit. Our intent, following after the original poster, was to discover all the possible ways that the background color of text marked as a line comment could be changed.

Our interests expanded after discovering a few answers via exploration. The reader can clearly observe in Figure 5.6(b), there are more options available in this

window than the ones stated in the user's question in Figure 5.6(a). For two new test suites we devised on our own, we expanded our focus beyond just exercising the Background-Color selection feature, to exercising the Text-Color selection feature.

We also found the question prompt to be a bit ambiguous upon deeper study of the question text. The original poster specifies that he wishes to change the background color of commented text within JEdit documents, but does *not* specify that the background color is the *only* thing he wishes to change. (*Indeed the original poster specified directly in the question that he "wants" to change "not just" the background but also the text color.)*

To answer all possible variants of what the user could want to accomplish from an answer to this question, we decided to attack this problem from three different angles. First we simply cover all possible ways the background color could be changed on its own, as well as all possible ways it could be changed alongside other changes that could *additionally* be made to the text. Since the user specified that they had already enacted changes to the text color before asking the question, we generated our second test suite without the background-color option in scope, but with text-color in its place, and got back ways this could be changed alongside other changes. Finally we created a test suite that included both the two coloration options amongst the other two options, in test cases.

This final suite consisted of all one, two, three, and four-way combinations of changing the state of background-color, text-color, font weight, or font-style (italicization) of text using the Style Editor window available from the Quick Settings menu of JEdit. This provided some very interesting output for our study, and it serves as the largest test suite to date that we have focused on.

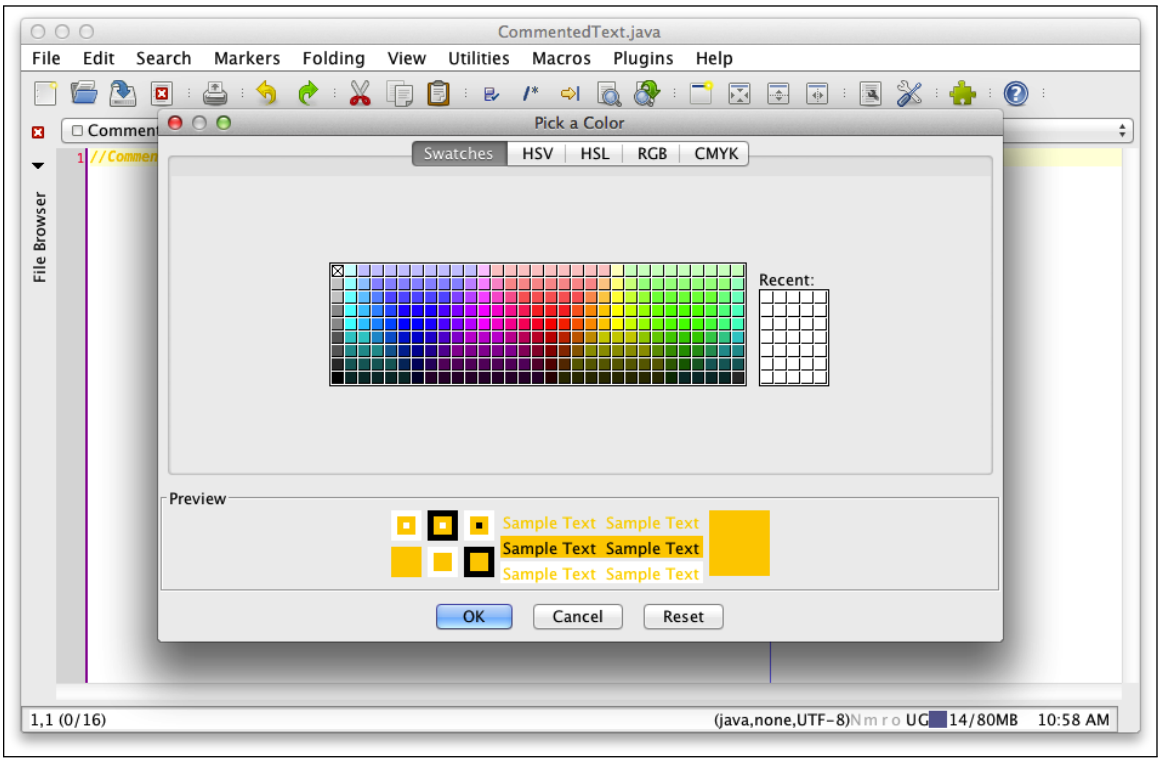


Figure 5.7: Snapshot of JEdit Color Selection Process

5.3.1 Variations in the Planned Goal

We will define legal variations of how our task might be executed: upon opening the Style Editor window:

The user may pick 1-4 of the following settings to change:

- The "Italic" setting checkbox.
- The "Bold" setting checkbox.
- The "Text Color" color picker button to the right of the Text Color label.
- The "Background color" color picker button (after first clicking the background-color setting checkbox to enable the button to the right of the label.

Having picked either of the latter two options, the user must immediately

- select a color swatch in the swatch palette that appears using the mouse or
- select a color swatch in the swatch palette using the keyboard arrow keys to move the select and the Space Bar to confirm (the plan for this task is to always choose the "gray" color swatch just below "white" in the top-left corner)

and following the choice, immediately click the close button of the "Select Color" window to return to the Style Editor window.

Choosing the same kind of style option from above twice is not admissible in any test case

The user must end the test case by clicking the final "Close" button at the bottom of the Style Editor window. All means considered, we are led to a total of 200 test cases that click the final close button at the end after changing the style of commented text.

5.3.2 Defense of Test Suite Correctness

5.3.2.1 Setup

Complexity of the amount of test cases we get back from this suite stems from the fact that they can be ordered in any way, and is also due to the fact that we must consider all one, two, three, and four-way combinations of the widgets in the Style Editor to achieve all possible methods of achieving the task. We'll explain why 200 is our target number for this test suite.

First, we enumerate all the options that the user can manipulate upon opening the Style Editor. They are

1. Select the checkbox labeled "Italics" to change the commented text to be italic-style [L]

2. Select the checkbox labeled “Bold” to change the font-weight to bold [B]
3. Change the color of the commented text by clicking the Text Color color picker and continuing to select a swatch via a mouse click [TCM]
4. ... or by selecting a swatch via the arrow keys and space bar [TCK]
5. Change color of the background behind the commented text by clicking both the checkbox in “Select Color” labeled “Background Color”, and clicking the color picker to the right to select a swatch using a mouse click [BCM]
6. ... or by selecting a swatch via the arrow keys and space bar. [BCK]

There are 6 total events available to select options. We’ll rfer to the final color changing options from here on as the “coloration options” of the kind Text Color and of the kind Background color, noting that there are two types of ways to select an option under each kind. We’ll refer to the first couple of event kinds as “simple options” that each have one way to activate the option.

Since we are not allowed to repeat any of the options, a simple way to enumerate all 200 is to count all the one-way, two-way, three-way, and four-way combinations of the 4 types of options available in the style editor window in a build-from-the-bottom fashion. We start by breaking the problem down into small parts.

Our formula starts out at:

$$6$$

For all the one way options we just enumerated that change the style. To start out *two-way*, we consider all the ways we can lead out of the previous step, step 1. We find this to be the simplest way of many to describe the process of mathematically proving our total 200 ways.

5.3.2.2 Finding All the Multi-way Combinations

We can next choose any option we have not picked, or immediately continue to the OK button and end the editing session. Upon choosing not to continue to the OK button after step 1, we continue to select a two-way option. If the test case starts for instance with selecting the italic checkbox [L], a simple option, it can be followed by 5 different possibilities: the bold checkbox [B] or one of the four of the different types of colorations stated above [TCM][TCK][BCM] or [BCK]. Likewise, if the test case starts with [B], just replace bold with italic in the former five options mentioned for italic to get 5 more types of possibilities.

For the colorations, say for a test case starting with [TCM], we can exit [TCM], and enter only 4 other remaining options. We have excluded TCK from the 5 options we have available we haven't chosen, because [TCK] and [TCM] would repeat the use of a Text Color option, which is not admissible in any test case.

We can likewise exit [TCK] and enter 4 other options, and we can exit [BCM] and [BCK] with 4 and 4 other options respectively, as we want to avoid the use of the same feature twice with any coloration.

In all 6 scenarios we have reviewed, there are 26 2-way combinations of widgets. Our formula now becomes:

$$6 + 26$$

And we now continue to 3-way test cases by continuing to build a longer test case beyond the test cases we now have. There are two cases we wish to consider amongst the set of length-2 test cases we now have.

Case 2.1: Let's take just those test cases that consist of two simple options [L] and [B]. There are two such members: ([L],[B]) or ([B], [L]). The third choice must be a

coloration, because we've run out of simple choices. We can exit either of these length two test cases and enter any of the four coloration options, creating 8 new 3-way test cases by doing so.

Our formula grows to $6 + 26 + 8+?+?$.

Case 2.2: Let's take just those length-2 test caes consisting of back-to-back colorations. Since they must be selected one per kind, there are 8 such ordered sequences

([TCM], [BCM]) ([TCK], [BCK])
([TCM], (BCK]) ([TCK], [BCM])
and their reverses.

Any of these 8 may be followed by one of the leftover simple choices that are yet unselected, [L] or [B].

Case 2.3: Now fo rhte remaining part dealing with the choice of one coloration and a simple choice. There are a larger number of these than in the previous cases: ([L], [TCM/K]) ([B],[TCM/K]), ([L],[BCM/K]) ([B],[BCM/K]), and the reverses of these pairs. In total there are 16 possibilities. No matter what, there are 3 leftover options that can follow this pairing, one of the simple options and the 2 types of a single kind of coloration.

Our formula as it stands now accounts for all three-element combinations of options from select color:

$$6 + 26 + (8 + 16 + 48).$$

Having already counted up all cases of 3-way possibilities makes it easier to move forward to calculate all the 4-way possibilities.

In all eight 3-way members of case 2.1 above, we are solely left with the choice of

picking one type of a single kind of coloration. Since there are two types, we simply multiply 8 by 2 to get the 4-way possibilities that start with [It] and [Bo]. Likewise in case 2.2 we get 16 more possibilities, because in all 16 3-way cases, we must pick a simple option for our fourth choice.

In case 2.3, we do some simple math. Since for the third choice we only had 3 options, and we must have picked either a simple option or one of two leftover coloration options to get the 48, exactly 1/3 of the members of case 2.3 must have ended with a simple option, thus leaving us with a choice to pick one of the types of coloration we didn't pick earlier as our fourth choice (Multiply 16 of these test cases by 2 to get these options). That leaves 2/3 of the members of 2.3 that must have ended with a coloration. These must be followed by the lone simple option left over. So our formula becomes

$$6 + 26 + (8 + 16 + 48) + (16 + 16 + 64) \text{ ways...}$$

$$= 32 + 72 + 96$$

$$= 200 \text{ ways.}$$

And we have the desired test case count.

Depicted in Figure 5.8 is a graph showing the flow of all test cases through a visual flowchart. The tasks all start with opening menu, Utilities, and progressing down until the choice can be made to pick one of the four options available in the style editor.

In table 5.1, we show how often certain rules were used for certain tasks, including the ones we depict in this walkthrough.

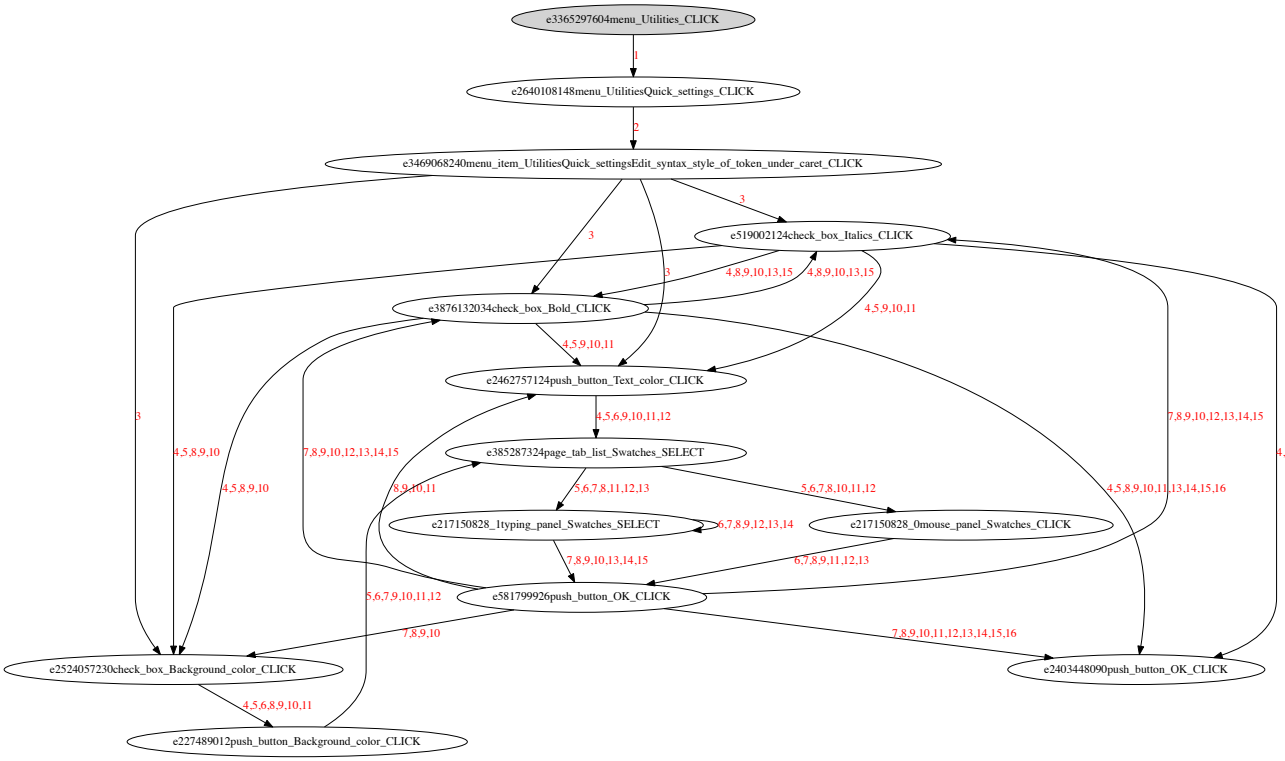


Figure 5.8: Constraints used in Each Test Case for Our Case Study

Table 5.1: Constraints used in Each Test Case for Our Case Study

Application	Category	Rules HPRT	Rules EFS	EFS Rule Types REQUIRE	EXCLUDE	ORDER	REPEAT	ATOMIC
Format Text	**M**	4	7	4	-	2	1	-
	MK	7	9	4	2	2	1	-
	MKT	7	10	4	3	2	1	-
Insert Hyperlink	**M**	6	7	5	-	2	-	-
	MK	8	9	5	2	2	-	-
	MKT	8	9	5	2	2	-	-
Absolute Value	**M**	6	8	5	-	2	1	-
	MK	8	10	5	2	2	1	-
	MKT	9	11	5	3	2	1	-
Insert Table	**M**	4	6	4	-	1	1	-
	MK	6	8	4	2	1	1	-
	MKT	7	9	4	3	1	1	-
Comment Indent		5	8	3	3	1	1	-
Search Options		6	7	5	1	1	-	-
Bold Center		6	8	4	2	1	1	-
Compile File		NA	6	2	2	1	1	-
Commented Text	**TC**	NA	2	1	1	-	-	-
	BG	NA	3	1	1	-	-	1
	FULL	NA	7	1	-	-	4	2
Four Paragraphs	**WD**	NA	12	4	3	1	-	4
	MD	NA	16	6	3	2	1	4

Chapter 6

Empirical Study

In this section we describe an empirical study to evaluate EventFlowSlicer. We utilize tasks that come from prior work on HPRT as well as new tasks that we have derived from Online forums.

The research questions that we ask in this study are:

RQ1: How does EFS compare with the HPRT test generation algorithm in terms of efficiency and effectiveness?

RQ2: Can we effectively generate test cases for the various classes of goals identified using EFS?

RQ3: What is the impact of each step of the EventFlowSlicer EFG reduction phase both on the EFG's complexity and overall generator runtime?

Experimental artifacts and results can be found online at `http://cse.unl.edu/~myra/artifacts/EventFlowSlicer/`

6.1 Study Subjects

For RQ1, we use tasks and test cases from the work of Swearngin et al. [6]. We were able to retrieve the EFG, GUI, and Constraints (Rules) artifacts down from a running online repository hosting artifacts from the study. These include test cases for four tasks in LibreOffice 3.4.3 suite, that utilize three of the LibreOffice applications modules, `Writer`, `Calc`, and `Impress`. In order to compare running time and study the effectiveness of our generator, we reuse the generator provided by Swearngin and run both generators on a total of 15 tasks, including 3 tasks created for three more applications to show generalization.

Three new tasks were tested on the following three platforms: `JEdit` version 5.1.0 [30], `DrJava` version 20140826-r5761 [31], and `TerpWord 3` (2003 release) [32]. For RQ3, we utilize an additional 9 test suites, to create a demonstration of 21 test suites total, that together demonstrate all the types of goals we set out to accomplish examples for in this thesis. The additional 9 test cases are generated using two of our new subjects, DrJava and JEdit, using new tasks.

Out of the additional 9 tasks we add to HPRT, 3 of these tasks, the tasks chosen for our first study, were informally imagined and formed for the purposes of comparing EFS with HPRT. RQ2 is concerned with finding real world applications for our generators uses, so 3 questions were taken from an online forum, StackOverflow.com, and broken down into 6 separate test suites in all for 2 that can be run on two of the new applications we review from earlier. We explain how these three questions were selected in the procedures (Section 6.3.2).

Table 6.1: Applications Under Study, Their Sources, and Resulting Size of Test Suites

Key	Goal	Category	Source	Test Suite Cases
1	**Libre Office Format Text M**	SDO	http://cse.unl.edu/~myra/artifacts/HPRT-2013	3
2	**Libre Office Format Text MK**	SDO	http://cse.unl.edu/~myra/artifacts/HPRT-2013	24
3	**Libre Office Format Text MKT**	SDO	http://cse.unl.edu/~myra/artifacts/HPRT-2013	81
4	**Libre Office Insert Hyperlink M**	SDO	http://cse.unl.edu/~myra/artifacts/HPRT-2013	2
5	**Libre Office Insert Hyperlink MK**	SDO	http://cse.unl.edu/~myra/artifacts/HPRT-2013	81
6	**Libre Office Insert Hyperlink MKT**	SDO	http://cse.unl.edu/~myra/artifacts/HPRT-2013	18
7	**Libre Office Absolute Value M**	SDO	http://cse.unl.edu/~myra/artifacts/HPRT-2013	4
8	**Libre Office Absolute Value MK (AVMK)**	SDO	http://cse.unl.edu/~myra/artifacts/HPRT-2013	32
9	**Libre Office Absolute Value MKT (AVMKT)**	SDO	http://cse.unl.edu/~myra/artifacts/HPRT-2013	72
10	**Libre Office Insert Table M (ITM)**	SDO	http://cse.unl.edu/~myra/artifacts/HPRT-2013	3
11	**Libre Office Insert Table MK (ITMK)**	SDO	http://cse.unl.edu/~myra/artifacts/HPRT-2013	12
12	**Libre Office Insert Table MKT (ITMKT)**	SDO	http://cse.unl.edu/~myra/artifacts/HPRT-2013	36
13	**TerpWord Bold Center (TWBC)**	SDO	*	8
14	**DrJava Search Options (DJSO)**	SFG	*	64
15	**JEdit Comment Indent (JECI)**	SDO	*	16
16	**DrJava Compile File (DJCF)**	SDO	http://stackoverflow.com/questions/26821167/how-to-determine-what-commands-drjava-is-executing	4
17	**JEdit Commented Text TC (JECTTC)**	SAG	http://stackoverflow.com/questions/27180136/highlight-comments-background-in-jedit	26
18	**JEdit Commented Text BC (JECTBC)**	SAG	http://stackoverflow.com/questions/27180136/highlight-comments-background-in-jedit	26
19	**JEdit Commented Text FULL (JECTF)**	SAG	http://stackoverflow.com/questions/27180136/highlight-comments-background-in-jedit	200
20	**JEdit Four Paragraphs N (JEFPN)**	SFG	http://stackoverflow.com/questions/33421810/finding-the-regular-expressions-search-in-jedit	114
21	**JEdit Four Paragraphs MD (JEFPMD)**	SFG	http://stackoverflow.com/questions/33421810/finding-the-regular-expressions-search-in-jedit	36

* Task has no online source

6.2 Study Metrics

First in RQ1, we wish to determine how effective we could be at using the HPRT generator to generate test cases, and the amount of precision we can reach at replicating the research of the Swearngin, et. al, in [6]. We collected data about the number of runs performed to generate output, the time it took to complete each run, and the factors that could have impacted the running time, such as number of nodes in the input EFG, and the size of the graph in terms of the edges. We also recorded and report on trends of running times across categories that we noticed, and also trends on running times across the different tasks we generated, and the total rate of accuracy we reached in replicating the test cases found on the study repository.

For this research question, we also measure the delta change from the size of the rule sets provided by the HPRT study to the size of our rulesets, noting the exact changes in the types of rules used, and provide observations about the dissimilarities, and what was required in the new generator to overcome obstacles.

After successfully replicating the generator's results, we wanted to find the runtime of the two applications running side by side. In order to provide a fair comparison, we executed both generators on the tasks input files, and measured the wall clock

runtime of the amount of time it took each generator to finish processing and write all its test case files to the file system. We took the times gathered from the replication study in RQ1, and compared these for each task to an average over 5 separate runs for the same task on the EFS generator. Since the HPRT generator required multiple runs of the generator to generate all test cases, we take the sum of all times it took each generation instance to process and generate test cases, minus the setup time between separate runs of the generator for different lengths, and add up the wall clock runtimes to get results for each task run using HPRT generator.

RQ2 allowed us more freedom to measure the strengths and weaknesses of our generator on problems we didn't know for sure our generator could solve. We note the procedure we took in selecting questions from online forums, and set up the study by measuring the number and varieties of constraints used to address each situation. We verified that we collected at least one task to meet each type of goal defined, and then after configuring the generator, verified whether each test suite validly represented all ways to reach the goal by manually examining each test case. Additionally we measure the lengths of each test suite, and the variety of constraints used, and the varieties of types of widgets involved in each old and new task we added.

For RQ3, we needed ways to understand the effectiveness of our reduction technique. The best measurement would seemingly be based on remaining edge count post reduction, since our reduction removes edges from EFG's. Since our reduction phases only remove edges and not nodes, we'll use the term graph edge-set size or **graph size** to refer to the size of the graph based on the number of edges. Our main question we aim to answer will then be this: Does the graph contain fewer edges after a reduction step is applied?

6.3 Study Procedures

6.3.1 Generator Comparison Study

All test cases were found at the URL's shown in Table 6.1. All test cases were given a key (1-21) to help identify them uniquely. Test cases 1-12 were used in our generator comparison study. We will now discuss the details of the generator comparison study. We use 12 tasks in this study, keyed 1-12 in Table 6.1 the data of which was downloaded from the online repository hosting the artifacts from [6].

The first research question insists that we first properly reproduce the results from HPRT. Our goal was to measure the efficiency (the runtime) and the effectiveness (the capabilities) of either product and compare them to see whether there were stark differences between the two in their performance. The data online, namely being an assorted collection of model assets and the rules files for 12 separate tasks, directly suited our purposes: we did not need the application since from the site we could gain the assets, the only artifacts actually needed to run the generators. Following this, we observed whether, given a certain application, a set of constraints, and a certain length parameter y if required: A) the time it took to run the HPRT and EFS test case generation algorithm, B) the capabilities of either generator to respond to expressive set of user input to generate the same results, and C) the amount of waiting time to see each generator compute complete results for the tester.

We have provided the reader with descriptions that serves as recipes, or exact sequences of steps, that can be used to complete the task in Figure 6.2, in the order they are listed in Table 6.1. HPRT presented different widget categories, but we group these together in this table of goals to focus specifically on the goal groups of tasks are attempting to accomplish perform, rather than on the model data. The tasks in Table 6.2 are also ordered by complexity. Unlike the descriptions provided in [6], our

Table 6.2: Table of Tasks Used in Our Study

Goal Name	**Description**
LibreOffice Writer Format Text (Struct. Diffs. Only)	In Document Viewer: Type the word "Chapter." Make the text typed bold. Make the text centered.
LibreOffice Writer Insert Hyperlink (Struct. Diffs. Only)	Open "Hyperlink" window. In Hyperlink window: type "www.amazin.com" in [Target] text box Type "Amazin" in [Text] text box. Confirm the changes, then click [OK]. Change the case of the text in the document editor to be uppercase.
LibreOffice Calc (Absolute Value) (Struct. Diff. Only)	Open "Function Wizard" window. In Function Wizard window: Click [Next] then type "-87" in [Number] text box. Confirm the changes, then shift the cell just edited one cell to the right. Turn off column and row headers in the formatted document.
LibreOffice Impress Insert Table (Struct. Diff. Only)	Open the "Table" window. In the Table window: Type "6" in the columns spinner text box. Confirm the changes, then add a new slide to the formatted presentation. Hide the task pane in the open presentation.
JEdit Comment Indent (Struct. Diff. Only)	(A pre-created .java file has been opened containing only the text "Edit the code here") Change the line of text into a java-source line comment. Shift the line of text one tab stop to the right.
DrJava (Search Options) (Same Functional Goal)	(A pre-created .java file has been opened containing only the text "Edit the code here") Open the "Find/Replace" panel in the Interactions pane. In the Find/Replace panel: Enter "Edit" in the [Find Next] box Enter "Replace" in the [Replace With] box. Search for and replace the word specified in the open document.
TerpWord Bold Center (Struct. Diff. Only)	In the Document Viewer: Type the text "Jonathan" using the cursor and keyboard. Make the text typed bold. Make the text typed centered in the document horizontally.
DrJava* Compile File (Struct. Diff. Only)	(A pre-created .java file has been opened, containing a no-argument java program.) Compile the document. Run the compiled class file with no arguments. The output pane displays the text "Hello World"
JEdit* CommentedText (Same Abstract Goal)	(A pre-created .java file has been opened containing the text "//Commented text.") Open the "Style Editor" window. In Style Editor window: Change appearance of all comments in current doc using any option(s). Confirm any changes, and exit the Style Editor. Style(s) of the commented text in the open document are changed.
JEdit* FourParagraphs (Same Functional Goal)	(A pre-created .java file has been opened containing four separated lines of filler content.) Open the "Search and Replace" window. In the Search and Replace window: Enter the JEdit regex ".+" in the [Search for] text box. Enter the JEdit regex *" $0 ."* Find and replace the four lines of content in the document, with new text wrapped in tags. All four paragraphs in the document are now wrapped in markup.

descriptions focus less on the names of the widgets, but instead convey a general feel for how complex the task is. Each *step* is given a single line. We take care to call out each of the elements in scope within the task using some part of each description. Due to space constraints, we do not list all the alternatives for each step, including the orderings that some steps may be performed in to complete the task properly. For most of these tasks where the effect is not obvious, the final line in the table

cell specifies the final effect expected within the application, and where that effect is expected to appear.

To answer questions about the comparability between HPRT and EFS, we setup this study to use the first 7 of the 10 tasks described in this table for the following reasons. The final two purposefully contained lots of events and edges in their models (the most out of any other task in our study), and thus were reserved for the feasibility study of our new generator rather than used in a comparison of our generator to an older version. The 7th task, though seemingly not too complex, seemed extraneous given the fact we already had a task for the new application DrJava prepared for use by HPRT and EFS.

6.3.2 New Goal Generation

Our three-step approach was planned out into the following three phases:

- Phase 1: Uncover questions by searching the website StackOverflow.com using StackOverflow's search query interface based on search criteria that help target our two applications in focus.
- Phase 2: Filter out a single question at a time according to specific criteria, allowing us to focus on questions we might be able to answer with our generator.
- Phase 3: Make formal our interpretation of each question as a "task" and its overall scope, and generate test cases that when replayed on the interface, achieve the goal the original poster stated they want to achieve.

The tasks we selected for this portion of our study are indicated and marked in Table 6.2.

6.3.2.1 Phase 1

We took questions from stack overflow based on a few criteria. We first sorted questions into a good-to-solve and not-good-to-solve column. Questions that we saw as not-good-to-solve were immediately filtered out of our selection.

A question would be put into not-good-to-solve column for a particular product because:

- the question regarded an installation issue that the user was having related to the product,
- the question regarded a problem compiling source code containing errors that the original poster simply wished to get fixed, (Dr. Java)
- the question regarded the problem of installing a plugin,
- the question regarded using a plugin relevant to the product, but the plugin was too obscure for me to understand its purpose in the interface, and thus whether we ourselves could answer the question using our generator,
- the only answers to the question regarded using a shell script that could be used to code their way into a solution using a system interpreter or a backdoor approach provided by the operating system,
- the original poster's question was limited to simply asking for a better alternative to the product itself, without referring to a specific problem with the interface,
- the question regarded an operating-system or JVM-related defect regarding the product, but not related to the interface, or
- the user's question was completely irrelevant to the product,

Phase 1 was conducted by searching through all questions returned by a given search query for the application, which we saved.

During our study we found it necessary to take multiple strides at finding the right search query. For each application, JEdit for example, we first attempted using a generic search term in StackOverflow's whole-site search widget: "**application-name** is:question", to pick up every single question that contained a reference to the applcation in the question text or answer text, and results pointing to questions only (not answers to questions). We found that this returned 240 results in total for the term "jedit is:question" JEdit application, and 196 results for "DrJava is:question".

Later on we found that questions returned by our initial search term had too many irrelevant results, and too many would quickly fall into the not-good-to-solve column. Rather than spend time sifting through too many negative results appearing anew, we set a cutoff. After we found the first 7 or so questions from our first query type fell into the final not-good criterion (completely irrelevant) for each of our applications, we tried another similar yet standardized search query for each application, that employed StackOverflow "tag queries". Upon further research in StackOverflow documentation, and when trying the new queries ourselves, we found that tag queries forced our search results to contain questions only, and also impose the constraint that a user logged into the site, marked the question with the same tag we chose to use in the search on our end - implying a mutual contract on search relevance. This proved useful to reducing question results page size by over half.

In the case of JEdit, the new search term was revised "[jedit]", and for DrJava, the search term became "[drjava]"). A total of **45** out of **101** questions returned by the new JEdit search query were sifted into the not-good-to-solve column for the JEdit application in accordance with our initial criteria, and **42** out of a total **106** questions returned by the DrJava query made it into the not-good-to-solve column for

DrJava-related questions. Results were sorted by date entered. The cutoff date for our Phase 1 study window when was the date of March 28, 2016. No new questions submitted to StackOverflow after that date were considered. All search results returned by our search on StackOverflow using these two terms, constitutes the entire corpus of data collected for this phase.

6.3.2.2 Phase 2

For the 56 questions on the topic of JEdit, and 64 questions on the topic of DrJava that made it into the good-to-solve column for the first round, we further reduced the column using new criteria after an initial pass. This line of criteria tailored questions to those that had to do more with our line of research in this study.

The new criteria were:

1. **All plugins necessary for the task to be completeable are freely available and easily accessible and installable.** Plugins to tools such as the JEdit text editor may change the interface in various ways, and without access to these plugins, we could not test the program in a way that would satisfy an answer to the question as the answer might suggest.

2. **Solutions could be effected using JEdit's current version.** Some original posters asked questions regarding features that were for features removed from the current version, or omitted or staged for later addition to JEdit's platform, which we could not tackle.

3. **Solutions requiring additional configuration resources were available to us during our study period.** More than a few questions we reviewed resulted in answers involving operating system API, remote file systems, or other resources that were not available to us during our study window.

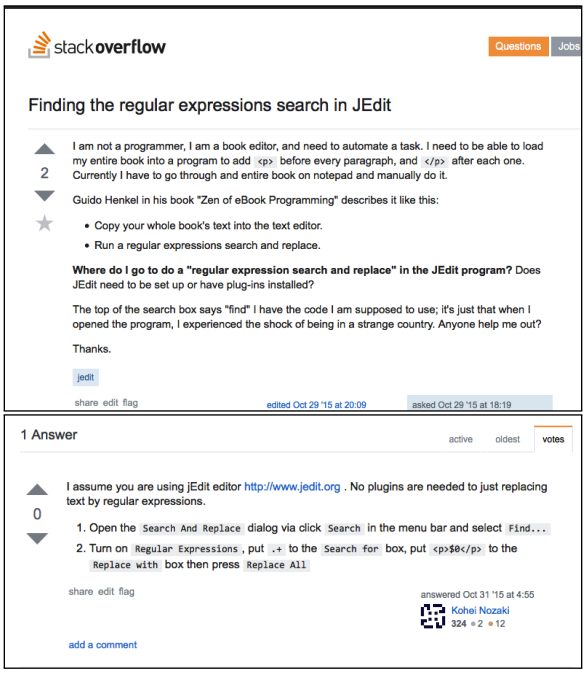


Figure 6.1: Example Stack Overflow Question and Answer

We made it 30% through the list before selecting some tasks to actually try to implement, based on how successful we predicted we'd be based on having studied more carefully the title of the question, and the question prompt.

6.3.2.3 Phase 3

In the end, we formalized a total of **four** tasks that we could further analyze and generate tests for, summarized in Table 6.2. Images of the original questions are given in figure 6.1.

6.3.3 Reduction Algorithm Efficiency

RQ3 poses a challenge to whether our reduction algorithms were effective in helping to reduce the size of the EFG's before the reduction phase. We collected the outcomes of test case generation reduction process midway through the computation process.

Table 6.3: Statistics from HPRT Replication Study

	Menus Only			Menus + Keyboards Shortcuts			MK + Toolbar Buttons		
	Nodes/ Edges	**Rules**	**Time**	**Nodes/ Edges**	**Rules**	**Time**	**Nodes/ Edges**	**Rules**	**Time**
FTW	9/29	4/7	8(s)	12/55	7/9	107(s)	15/109	7/10	187(min)
IHW	9/37	6/7	7(s)	11/61	8/9	35(s)	13/99	8/9	295(s)
AVC	11/34	6/8	16(s)	14/57	8/10	280(s)	16/82	9/11	71(min)
ITI	7/17	4/6	4(s)	9/31	6/8	23(s)	11/53	7/9	45(s)

These outcomes are reduced EFG graphs, one produced after each reduction of the EFG via our 3 reductions outlined in Chapter 4.

To measure the efficiency of our reduction algorithms, for each task we ran the EFS generator one time. The output of the generator is defined such that for every run, statistics are reported about various attributes of the graph: we went back to older output from the other two studies, and then collected edge count and cyclomatic number from the output files generated. We ran these through statistical analysis and present in the results, bar charts summarizing the results of the reductions, and also collected the mean amount of reduction of the EFG each reducer accomplished across all 21 test cases. In addition, since cycles have been known to cause test suite blowup, we report on the reduction of cyclomatic complexity of each edge reduced EFG.

6.4 Results

6.4.1 RQ1

Research Question 1 How does EFS compare with the HPRT test generation algorithm in terms of efficiency and effectiveness?

Table 6.4 shows the lengths all of the test cases that we generated using Event-

Table 6.4: Test Cases Lengths of All EventFlowSlicer Test Suites

Task	Cat.	Total TC'S	breakdown (L) length/ (C) count							average length	unique task tests
Format Text M	SDO	3	L	**10**	-	-	-	-	-	3	3
			C	3	-	-	-	-	-		
MK	SDO	24	L	**6**	**7**	**8**	**9**	**10**	-	5.5	24
			C	3	6	6	6	3	-		
MKT	SDO	81	L	**4**	**5**	**6**	**7**	**8**	**9**	**10**	5.5	81
			C	12	6	18	21	12	9	3		
Insert Hyperlink M	SDO	2	L	**9**	-	-	-	-	-	-	9	2
			C	2	-	-	-	-	-	-		
MK	SDO	8	L	**6**	**7**	**8**	**9**	-	-	-	7.5	8
			C	2	2	2	2	-	-	-		
MKT	SDO	18	L	**6**	**7**	**8**	**9**	-	-	-	7	18
			C	8	4	4	2	-	-	-		
Absolute Value M	SDO	3	L	**10**	-	-	-	-	-	-	10	4
			C	3	-	-	-	-	-	-		
MK	SDO	32	L	**9**	**10**	**11**	**12**	-	-	-	10.5	32
			C	4	12	12	4	-	-	-		
MKT	SDO	72	L	**7**	**8**	**9**	**10**	**11**	**12**	-	9.5	72
			C	8	12	12	20	16	4	-		
Insert Table M	SDO	3	L	**8**	-	-	-	-	-	-	8.6	3
			C	3	-	-	-	-	-	-		
MK	SDO	12	L	**6**	**7**	**8**	-	-	-	-	7	12
			C	3	6	3	-	-	-	-		
MKT	SDO	36	L	**5**	**6**	**7**	**8**	-	-	-	6.3	36
			C	3	15	12	3	-	-	-		
Comment Indent	SDO	16	L	**3**	**4**	**5**	**6**	**7**	**8**	-	5.5	16
			C	2	2	4	4	2	2	-		
Search Options	SFG	64	L	**7**	**8**	**9**	**10**	**11**	-	-	8.6	64
			C	2	8	18	24	12	-	-		
Bold Center	SDO	8	L	**8**	**9**	**11**	**12**	-	-	-	10	8
			C	2	2	2	2	-	-	-		
Compile File	SDO	4	L	**2**	**3**	**4**	-	-	-	-	3	4
			C	1	2	1	-	-	-	-		
Commented Text N	SAG	26	L	**5**	**6**	**9**	**10**	**11**	-	-	9.5	250
			C	2	2	4	10	8	-	-		
BG	SAG	26	L	**5**	**6**	**9**	**10**	**11**	**12**	-	10.1	
			C	2	2	1	5	10	6	-		
FULL	SAG	200	L	**5**	**6**	**8**	**9**	**10**	**11**	**12**	14.2	
			C	2	2	1	7	15	15	6		
				13	**14**	**15**	**16**	**17**	-	-		
				3	16	50	59	24	-	-		
Four Paragraphs N	SFG	114	L	**9**	**13**	**15**	**16**	**18**	-	-	15.9	150
			C	6	6	48	6	48	-	-		
MD	SFG	36	L	**11**	**19**	**20**	**25**	**26**	-	-	21.4	
			C	4	4	12	4	12	-	-		

FlowSlicer for this research question. Beneath each "length" bin we record the number of test cases that fit that category.

This table points out, for the first 12 test cases, the fact that for some categories, the HPRT test case generator required passing not a single, but multiple length parameters to the generator, to generate all output specified for the tasks specified.

It is important to point out in our discussion a benefit/drawback tradeoff of HPRT related to this. The tester needs accurate information about the length of each test case to verify whether or not the whole test suite initially targeted was generated after

a single run. Given some oracle or world knowledge about the length of each test case is available, the specifications of the generator ensure that all test cases of each length specified that satisfy constraints are outputted. Thus with the right information, we can generate the right test cases.

The report in [6] gave this number to be expected for each task, and the data on the repository confirmed most of our efforts. Once this number was achieved, there was no need to continue to run the generator for any more parameter inputs.

We used an XML file differencing tool to check the files we generated, and found that we were successful able to replicate a total of **292 of the 295** test cases available for replication on the HPRT study repository. The test suite we generated contained an extra 3 test cases. In light of the perceived error from earlier, we proceeded to compare these new files against the ones we had generated. We found that generating the files missing from the FT MKT test suite could be done using both generators, and were able to prove that these files could indeed replace the corrupted files in the repository, by observing the task itself and verifying each of the extra three as ones that address the leftover needs within a completed test suite that the corrupt three would have addressed had they held valid data.

6.4.1.1 Constraint Deltas

We wanted to observe whether changes to constraints were complex or sizable for the tasks studied in HPRT or for the tasks we added to the study as new application tasks. Formally, our research hypothesis for this particular subquestion of question 1, was: *A significant amount of rules (10% of the original constraint space) must be modified or created to originate new constraints for use by the new generator when generating the same test suites created the old generator generated.*

We found that on average, 30% of the constraint space was modified. 3 rules

Table 6.5: Constraints Used for EFS Tasks taken from HPRT Study

Application	Category	Addl. Required Rules for EventFlowSlicer MI = Menu Item, TBB = Toolbar Button KS = Keyboard Shortuct
Format Text M	SDO	Select All MI, Bold List Item, Align Center MI
Format Text MK	SDO	(Select All MI OR Select All KS) (Align Center MI OR Align Center TBB)
Format Text MKT	SDO	(Select All MI OR Select All KS OR Select All TBB), (Bold TBB or Bold List Item), (Centered TBB or Centered MI or Centered KS)
Insert Hyperlink M	SDO	UPPERCASE MI
Insert Hyperlink MK	SDO	(UPPERCASE MI or UPPERCSASE KS)
Insert Hyperlink MKT	SDO	(UPPERCASE MI or UPPERCASE KS or UPPERCASE TBB)
Absolute Value M	SDO	Column & Row Headers MI, Shift Cells Right Radio
Absolute Value MK	SDO	(Show Headers MI or Show Headers KS), Shift Cells Right Radio
Absolute Value MKT	SDO	(Show Headers MI or Show Headers KS), (Shift Cells Right Radio or Shift Cells Right TBB)
Insert Table M	SDO	Insert Slide MI, Show Task Pane MI
Insert Table MK	SDO	(Insert Slide MI or Insert Slide KS), (Show Task Pane MI or Show Task Pane KS)
Insert Table MKT	SDO	(Insert Slide MI or Insert Slide KS, or Insert Slide Dropdown List) (Show Task Pane MI or Show Task Pane KS)

Table 6.6: Constraints Used for New EFS Tasks in Study 2

Application	Category	Rules (MI = Menu Item, TBB = Toolbar Button, CHK = Checkbox Item)			
		REQUIRED	EXCLUSION	ORDER	REPEAT
TW BC HPRT	SDO	Document Text Area, Select All MI	(Bold List Item XOR Bold TBB), (Align Center MI XOR Align Center TBB)	Document Text Area, Select All MI, (Format Menu, Align Center TBB, Bold TBB)	Format Menu
EFS (Additional)	SDO	(Bold List Item OR Bold TBB), (Align Center MI OR Align Center TBB)	–	–	–
JE CI HPRT	SDO	–	(Line Comment MI XOR Line Comment TBB), (Shift Right MI XOR Shift Right TBB), (Select All MI XOR Select All TBB)	(Select All MI and Select All Toolbar Button TBB), (Shift Right MI and TBB), (Line Comment Menu Item and ... Toolbar Button)	Edit Menu
EFS (Additional)	SDO	(Line Comment MI or Line Comment TBB), (Shift Right MI or Shift Right TBB)	–	–	–
DJ SO HPRT	SFG	Find Next Text Box, Replace With Text Box, Find Next Button, Replace Button	Find/Replace MI XOR Find TBB	(Find/Replace Menu Item, Find TBB), (Find Next Text Box, Replace With TBB), (No Comments CHK, Whole Word CHK, Match Case CHK), Find Next Button, Replace Button	–
EFS (Additional)	SFG	(Find/Replace MI OR Find TBB)	–	–	–

needed to be added for every 10 already specified in the file. In the end we need to make sizable amounts of changes to the constraints set, however noting that the only types of constraints we changed, across 15 test cases with varying amounts of widgets, was using the (Mutually) Required rule, requiring that one of a set always be present in an output test case. Our hypothesis of less than 10% modification was proven false.

We depict in Table 6.5 and depict in Table 6.6 the new rules defined in our study of three new applications. For each we devised a set of new constraints. We show the difference between HPRT (odd columns) and additional constraints we had to add to EFS to get the same result. As is shown, the removal of the length parameter required that we specify to the generator that a set widgets be included in the output. Cutting off length of a test case, maintaining other rules proved to be effective in the test cases studied in HPRT. We hypothesize that this technique of simply constraining length without constraining other parameters will not work for more complicated tasks. At this time we cannot attempt to prove this to be true using our current data set.

In Table 6.6, we also demonstrate our ability to generate new test cases using the old generator. The time it took to devise these new constraints was not long (less than one day to get the right test suite). We attribute this to the tight mapping of the HPRT constraint sets to both real and common needs of the tester: it was straightforward to determine how to set up the inputs. The document text area needed to have text inserted, then the Select All button needed to be clicked and followed by just one method of centering the text or bolding – all of which easily broken down into exclusion, order, and required rules each maintaining part of the whole set of constraints necessary. We found we were able to generate the right number of test suites using the HPRT generator: we were able to, enter, centered and bold text it in all possible ways within the scope of widgets selected for this exercise. We replicated our success in the second and third instances, a task that commented some predefined

text and shifted it right in the document using JEdit, and a task that executed a search, find and replace operation on an open file using DrJava.

We quickly turned to EFS, and realized we again only needed to add 3, 1, and 2 rules to the constraint pool, which consisted of 22 rules total before adding new rules, for each task we had generated to create exactly the same outputs as before using the EFS strategy instead. Since the constraints files are mostly backward compatible we used EFS capture to generate the constraints, and feed hand modified versions to both generators.

6.4.1.2 Running Times

Further results of the replication study are explained in Table 6.3 which presents the amount of time the application under test spent running for each task we fed to the generator. To generate all the output the author provided, there are three parameters we must provide to their system: testing target (class files and system-specific inputs), testing model asset artifacts (GUI, EFG, and Rules file), and test case length y. To generate all test cases, the generator is repeatedly run to provide test cases of various lengths. There were 3 test cases that filled space in the repository under FT MKT, but contained invalid event data that did not corroborate with events in the EFG file or widgets in the GUI file for FT MKT. We posit these discrepancies were due to repository upload error. Observations were made on the running time differences between our generator and another generator like it in the same line of research, making HPRT a perfect candidate for such a comparison. Our goal was initially to improve upon the running time of this generator. Our research hypothesis was
The running times on the amount of time EFS and EventFlowSlicer spent running and generating test cases was the same on average.

We measured the delta change in running time from EFS generator runs to HPRT

Table 6.7: Cumulative Run Times: Sorted by Task Key (left) Sorted by EFS time (right)

Key	Goal	Category	EFS time(s)	HPRT time(s)	Diff	% Reduct
1	FTM	SDO	9.2	8	-1.2	-13.0%*
2	FTMK	SDO	12.6	107	94.4	88.2%
3	FTMKT	SDO	19	11257	11238	99.8%
4	IHM	SDO	6.6	7	0.4	5.7%
5	IHMK	SDO	9.2	35	25.8	73.7%
6	IHMKT	SDO	10.4	295	284.6	96.5%
7	AVM	SDO	10	16	6	37.5%
8	AVMK	SDO	12	280	268	95.7%
9	AVMKT	SDO	18	4254	4236	99.6%
10	ITM	SDO	7.2	4	-3.2	-44.4%*
11	ITMK	SDO	8	23	15	65.2%
12	ITMKT	SDO	15	45	30	66.7%
13	TWBC	SDO	8.0	106	98.0	92.5%
14	DJSO	SFG	14.8	261	246.2	94.3%
15	JECI	SDO	5	45	40	88.9%
16	DJCF	SDO	4.8	N/A	N/A	N/A
17	CTN	SAG	10.8	N/A	N/A	N/A
18	CTBG	SAG	11.6	N/A	N/A	N/A
19	CTF	SAG	19.6	N/A	N/A	N/A
20	FPN	SFG	15.8	N/A	N/A	N/A
21	FPMD	SFG	10.4	N/A	N/A	N/A
Statistics						
Average pct. Reduction All Instances				63.2%		
Containing Test Cases w/ Lengths >9				92.5%		

Key	Goal	Category	EFS time(s)	HPRT time(s)	Max TCLen	Nodes	Edges
16	DJ CF	SDO	4.8	N/A	4	5	17
15	JE CI	SDO	5	45	8	9	41
4	IH M	SDO	6.6	7	9	9	37
10	IT M	SDO	7.2	4	8	7	17
11	IT MK	SDO	8	23	8	9	31
13	TW BC	SDO	8.0	106	12	12	61
1	FT M	SDO	9.2	8	10	9	29
5	IH MK	SDO	9.2	35	9	11	61
7	AV M	SDO	10	16	10	11	34
6	IH MKT	SDO	10.4	295	8	11	99
21	FP MD	SFG	10.4	N/A	26	12	35
17	JE CTTC	SAG	10.8	N/A	11	11	35
18	JE CTBC	SAG	11.6	N/A	12	12	44
8	AV MK	SDO	12	280	12	14	57
2	FT MK	SDO	12.6	107	10	12	55
14	DJ SO	SFG	14.8	261	10	11	90
12	IT MKT	SDO	15	45	8	11	53
20	FP WD	SFG	15.8	N/A	18	12	113
9	AV MKT	SDO	18	4254	12	16	82
3	FT MKT	SDO	19	11257	10	12	55
19	JE CTF	SAG	19.6	N/A	17	13	51

* **For some very small test suites, the HPRT generator beat EFS by less than 2 seconds.**

generator runs. We saw a net 62.3% reduction in running time on average. We have mentioned in the Algorithms section how and why EFS outranks HPRT generator in terms of efficiency. For test suites that contained test cases of lengths 9 and above, our generator outperformed HPRT via a 92.5% reduction in runtime on average across these instances. The focus on these instances widens the gap: EFS runs on HPRT's longest test suites, and takes seconds to finish where HPRT took hours.

We depict statistics on the left in Figure 6.7 to show on average the reductions in time to completion for HPRT generator to complete its execution versus the EFS generator. Overall we saw an average of 63.1% reduction of runtime when comparing HPRT results, when we include all the reduction percentages at once.

There were cases where the HPRT generator ran faster than DFGen generator, in the cases of Format Text M, and Insert Table M. In these two cases, the HPRT generator finished 1.2s and 3.2s faster than DFGen could across 5 runs.

The DFGen algorithm uses a different technique than the HPRT generator to produce valid test cases adhering to constraints. The algorithm Memon designed,

which builds an EFG based on the widgets available in a multi-window environment, implies in its core pseudocode, a crucial corollary. We need this corollary to understand why the running times between HPRT and EFS generator are different.

For a simple interface that has buttons, say one single window interface, and one with say one single button : suppose we want to add a single button to the one-windowed button. Each widget like that button we add to that interface gains at least one edge leading from the old button to the new one, and from the new button to the old one, provided that neither button closes their shared window. What this means is that the number of edges is always growing with the number of nodes, and that additional widgets we add, if they share only a few windows amongst them, continuously grow the size of the edges, and if no windows are closed, grow the size of the edge set at an exponential rate.

Adding one button to the one-button window creates an edge set of size 2, but adding another creates one of size 4, another button creates 9 edges, the next 16, the next 32. It should become obvious to the reader after understanding this simple corollary why EFG's become so complex after 9 or so buttons exist in a window's interface. Memon's method extends beyond single window interfaces, and thus the EFG construct simply becomes more complicated from there. Extensions we've made to GUITAR that build EFG's from the ground up retain this basic property of same-window edges between widgets.

Observe the results in Table 6.7 on the right, that sort the results by the EFS running time, compare that to the order of how the last 7 tasks in Table 6.2, the tasks table. For HPRT, larger more complex test suite EFS means longer running time. For EFS, this is still true. Larger, more complex test suites have more nodes, which means paths to traverse are longer. But with tree pruning, less of these paths must be traversed to get the right answer.

Table 6.8: EventFlowSlicer: Widgets used per Task and Snapshot of JEdit FPWD

App and Task	Category	Total Events	EFS Rules	Breakdown									Test Cases
				PBUSTHTONS	MIETNEUMS	TBEOXXTES	CBOOMXBEOS	CBHOEXCEKS	RBAUDTITOON	FLLIASTT	KCEOYMMAND		
FTM	SDO	9	7	1	3	1	-	-	-	1	-		3
FTMK	SDO	12	9	1	3	1	-	-		1	3		24
FTMKT	SDO	15	10	4	3	1	-	-		1	3		81
IHM	SDO	9	7	2	2	2	-						2
IHMK	SDO	11	9	2	2	2	-				2		8
IHMKT	SDO	13	9	4	2	2	-				2		18
AVM	SDO	11	8	3	3	1	-		1				4
AVMK	SDO	14	10	3	3	1	-		1		3		32
AVMKT	SDO	16	11	5	3	1	-		1		3		72
IHM	SDO	7	6	1	3	1	-						3
IHMK	SDO	9	8	1	3	1	-						12
IHMKT	SDO	11	9	1	3	1	2				2		36
TWBC	SDO	8	2	2	3	1	-			1			8
DJSO	SFG	11	7	4	1	2	-	3					64
DJCF	SDO	9	7	3	2	-	1						4
JECI	SAG	9	8	3	3	-	-						16
JECT	SDO	13	7	6	1		-	3			1		124
JEFPN	SFG	12	12	5	1	2		1	1				114
JEFPMD	SFG	12	13	1	7	2	1	1					36

From our results, we observe if we follow the corollary described in our factors in efficiency discussion (Section 6.4.1.2), that problems incurred in running time should be reduced on average across all applications, by not running traversals using EFG's with more edges and using a length parameter to bound the traversal(as would have been done by HPRT).

Summary of RQ1: EventFlowSlicer is as effective as HPRT, and more efficient at traversing paths.

6.4.2 RQ2

Research Question 2 Can we effectively generate test cases for the various classes of goals identified using EFS?

In the introduction, we identified were three goals that testers aim for when generating all possible ways to perform a tasks. *Structural Differences Only, Same*

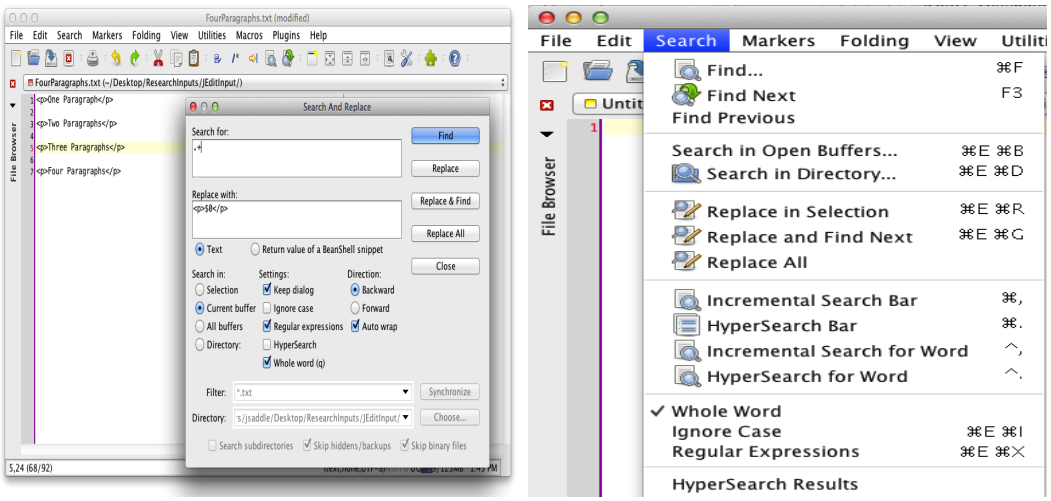


Figure 6.2: Variants of the JEdit FPWD task

Functional Goal, Same Abstract Goal. We were successful at finding real world instances that fit each of these goals according to the method described above. We proceeded to generate test suites for each of these and report on aspects of the output in the tables that follow. We successfully generated test cases from all the task specifications we defined in Table 6.2. We now show how our goals that deviate from the norms presented in HPRT extend into Same Functional Goal and Abstract Goal tasks.

The Commented Text task bears some explanation. Though we specify the user can change anywhere fom 1 - 4 options presented, the user may only select a single color from the 256-color palettes presented midway through each variant of the task. Even still there are yet 200 test cases that came from this one instance. We have broken down the Commented Text goal breaks down into three variants. The TC and BC variants of JEdit Comment Text restrict the types of options to the set of options $\{[TextColor], [Italics], and [Bold]\}$, and $\{[BackgroundColor], [Italics], and [Bold]\}$ respectively, while, the F (FULL) variant allows the selection of up to all four style choices at once. The number may seem quite high: it is because there are two distinct

ways select text color and two ways to select background color (via mouse and via keyboard), plus the fact that all orders of 1-4 selections are allowed, that make this our largest test suite in consideration.

It should be clear now why this goal is defined "abstract": the application will reside in a number of states depending on what options are selected. As for the Four Paragraphs task and Search Options task, we turn to the data in Table 6.8 and a figure depicting the FPN task 6.2. This table helps us understand the types of widgets that are used to find and replace text. Completing the task requires at least two completely separate routes through the JEdit Searching window to cover a complete the test suite. Repeatedly clicking the *Find* and *Replace* buttons in alternation is one way requiring 8 distinct steps. However, the same effect of doing that is discoverable by clicking the *Find* button, then the *Replace&Find* button three times, and clicking the replace button, leading to an alternative 5-step process. Moreover each of the preceding ways to accomplish can be preempted by clicking the [*AutoWrap*] and [*Backward*] options. This may reverse the process of searching through the document, but the end result is still the same, and thus the task defined in 6.2 of wrapping all four paragraphs in markup is still completed. This task has many means of accomplishing the same functional goal of wrapping four paragraphs of text in specified markup. It's clear to see from the table above the figure that a variety of types, and not just a single string of usual types of widgets, were used both in the Window Dominant varian(marked WD) of this task. The Menu Item Dominant (MD) is marked that way because it utilizes more menu items as opposed to options from the Searching window of JEdit to perform the meat of the task. Much of the options are available in the portion of the search menu that are also available in the Searching window, and we wanted to point this out as a separate task rather than combining the two. The end goal is

still the same, yet the total change in the types of widgets used in both tasks can be derived from the chart in 6.8.

Summary of RQ2: The answer to research question 2 has been answered. We do support the creation of tasks that exemplify goals with test cases reaching beyond structural differences only.

6.4.3 RQ3

Research Question 3 What is the impact of each step of the EventFlowSlicer EFG reduction phase both on the EFG's complexity and overall generator runtime?

Certain edges in every EFG get removed by our reductions. These are edges the EFG generator preserves during its construction, but that are not useful in real user tasks: they serve as the root cause in the generation of event sequences in test cases that progress too slowly or not at all toward realistic user goals that are feasible in the real world.

We evaluated our approach of removing these unhelpful edges by measuring the "size" of our EFG (the number of edges remaining) after a reduction, and comparing that size to a base size: the size of the **"minimal" test case flow graph**, containing only the edges that were used in the test suites we generate for the task in question. This graph is specific to the task, thus we constructed 21 minimal test case flow graphs, one for each task. This "minimal EFG" tells us how many edges of the original EFG were removed out of the total edges that were *possible to remove* without breaking any test cases. If the ratio of the minimal TCG size to the reduced EFG size = 1, then we have reduced the graph as much as it can be reduced. Otherwise, if less than 1, then we know there were more edges remaining that could have been removed. We derive a ratio from this comparison for each task. Then we derived a total reduced ratio.

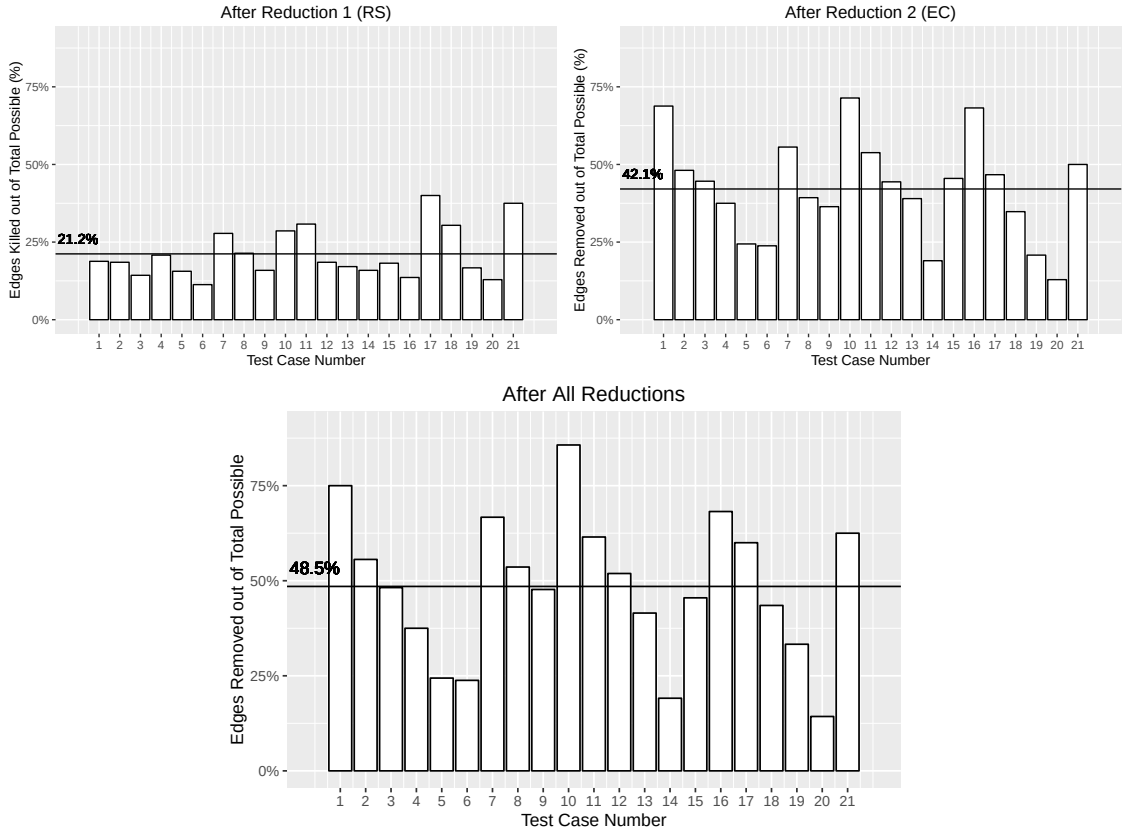


Figure 6.3: EFG Size Reductions out of Maximum Possible (black bar indicates the mean)

We explain how this was calculated using the following formula. Let E be the number of edges in the input EFG, let E_{Red} be the number of edges remaining following one or more reductions, and let E_{Min} be the number of edges in the minimal EFG.

Then the percentage of edges reduced out of total possible is:

$$PCT = \frac{E - E_{Red}}{E - E_{Min}}$$

We gathered statistics for the overall reduction of the size of the input graph, and we depict the percentage reduction for each of our test case instances from key 1-21 in Figure 6.3. On average, the input graphs we fed to the generators started out at

2.42 times of the size of the minimal flow graph on average. By the end of the third reduction, we had reduced each graph to 1.83 times the size of the minimal flow graph, our largest reduction yielding a reduced graph 1.14 times the size of its minimal graph. Out of total possible edges we ended up removing 48% out of the total number of edges that could possibly be removed.

From these results, we can see that our reduced graphs often contained more than 1.5 times the number of edges than were necessary to generate test cases. It often became a task left to our deep traversal procedure and the constraint-based pruning to help generate the right test cases that each followed the right paths along the edges, and to help end generation quickly.

The state following the *MenuToNonMenuChild* reduction (reduction 2) saw the greatest drop in edges on average across all 21 of our instances, forcing the mean reduction overall up to be 11 points higher (from 21.2% to 42.1%) by the end of the reduction step, as shown in Figure 6.3. The most edges any one graph lost from this reduction step, which removed edges leading away from menus and combo box revealers to items elsewhere on the interface that are not their children (to prevent useless clicking of menus), was 17, the mode was 1 edge, and the median was 4.

The EFG reduction strategy worked better for some application instances than it did for others. On average we only removed 45% out of the total possible egdes we could have removed in order to create new test cases (our maximum was 75% for instance Insert Table M from the HPRT study. That's an effectively moderate number of edges removed per instance, perhaps high enough to promise a decent speedup using a test suite reduction algorithm.

We did not attempt to see whether an actual speedup could be gained however. The EFS generator was designed with reductions built in. In the design process for desigining the EFS generator's specifications, it was decided that the reductions when

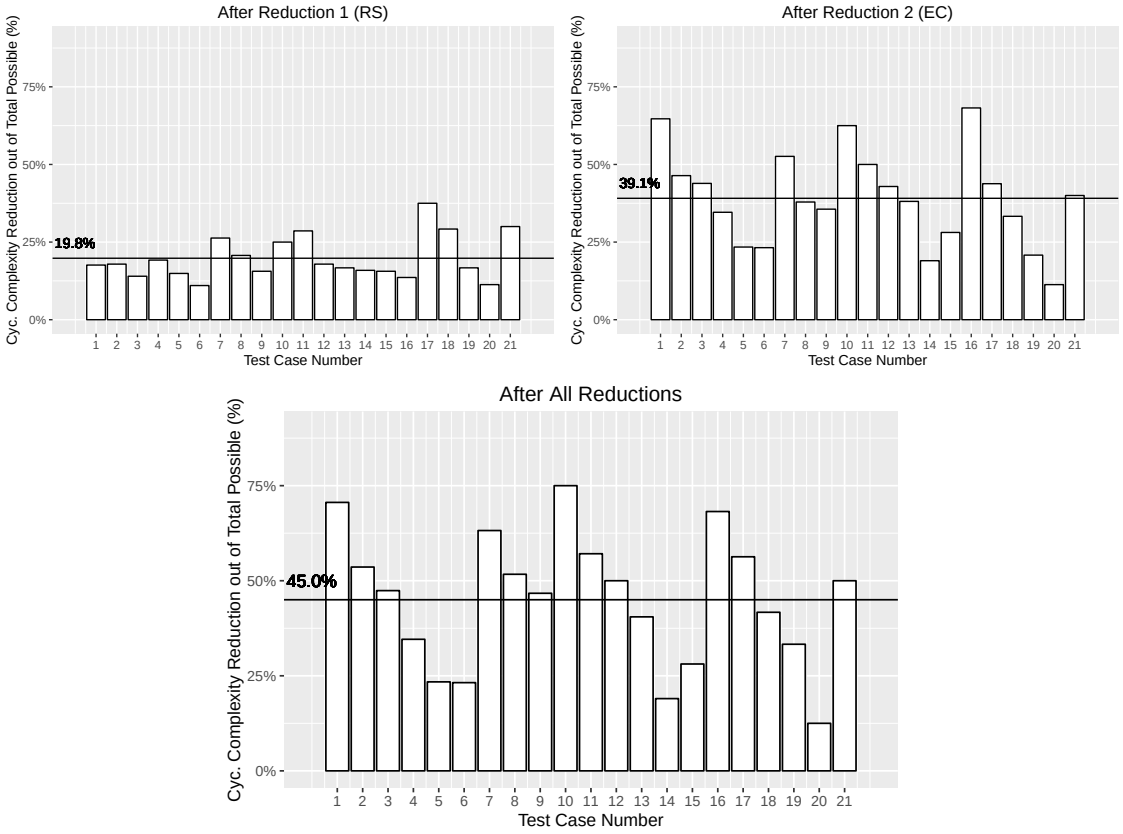


Figure 6.4: EFG Cyclomatic Complexity Reductions out of Maximum Possible (black bar indicates the mean)

used would help assist the tester in defining fewer constraints for test cases. Thus we anticipated that when reductions are included versus not included, the amount of constraints from our constraints pool that would remain necessary for the user to specify would change. After running multiple tests on multiple instances, we quickly found this to be true. Thus we assume that for all 21 instances, the constraints must be reconfigured (or at least re-verified) to generate the correct amount and quality of test cases we generated when the reductions are not in place. Due to the time allotted to this section of the study, we have chosen to leave the reconfiguration of new constraints in order to perform a fair comparison to a "no-reduction" EFS, up to our future studies.

6.4.3.1 Cyclomatic Complexity

For each measurement we took on graph size for each generation instance, we recorded measurement of the cyclomatic complexity in parallel with that measurement, to see how well cyclomatic complexity varied across each reduction. In accordance with our minimal test case flow graph measurements for *edges*, we took time to gather the cyclomatic complexities of all the minimal test case flow graphs, and we modified the formula above to read C rather than E to derive the minimum cyclomatic complexity measure in each test case graphs, and compare that to the complexity of each stage of our reductions. Note that the cyclomatic complexity formula we used takes into account the number of nodes, the number of edges, and the strongly connected components detected per graph. It has traditionally been used to depict the complexity of a graph network that may contain many cycles. The number of nodes and strongly connected components (inner cycles) add to the complexity, while the number of edges that don't form new SCC's reduce it.

Then the percentage of edges reduced out of total possible is:

$$PCT = \frac{E - E_{Red}}{E - E_{Min}}$$

Depicted in Figure 6.4 are the results. The cyclomatic complexity of original graphs were nearly 4.65 times that of their minimal counterparts on average, our maximum readout topping out at over 10 times the complexity of the minimal EFG for that instance. At maximum our reductions removed up to 75% of the possible units of complexity that could be removed, and the algortihm reduced cyclomatic complexity by 45.0% possible units on average.

6.4.4 Statistical Correlations

Were there trends that we could report on between graph size and running time?

We drew statistical conclusions on whether there are notable and significant trends connecting graph size cyclomatic complexity to running time. After finding out how similar the trends between graph complexity and running time, we dug deeper to find whether the two trends pose a noticeable difference from one another – thus comparing the two correlations, in order that we might see whether one of these predictors could be suitable as a substitute for the other when predicting running time.

We obtained a Pearson correlation between the running times from all 21 generations and various graph sizes obtained from all 21 input graphs from our previous studies. Since we were able to run HPRT on 15 of these examples, the correlations for HPRT time reflect trends across 15 of the examples – keys 1-15 in 6.1. We ran a 2-tailed statistical power test of significance to determine whether we could find any statistical meaning behind the data.

Table 6.9 shows the results of statistics we gathered on these graphs, and also provides data on relevant data collected across all suites. Our results indicate a few trends that were expected, and a few unexpected trends: the size of the graph correlates with the running time of the generator: as the number of edges in an input graph decreases, so does the running time of the generator ($r_{HPRT} = .583, p = .022$)($r_{EFS} = .583, p = .005$). Since $p < .05$, our results indicate that both edge size and cyclomatic complexity will reliably predict trends for both generators' running times. The fact that running time follows the rise of graph size is expected and confirms previous findings in the literature, [10,22] while our findings regarding cyclomatic complexity on inputs for these generators are new. Since the correlations are not equivalent the same, We believe it might be interesting to see whether the two relationships significantly

differ in their power to predict running time. Due to our time dedicated to revealing statistics over our data sets being limited, we leave the discovery of differences in their ability to predict running time to our work in the future.

Both generators use the same types of inputs, and we found that the running time of either generator can be predicted equally well by either the edge count or the cyclomatic complexity. EFG size and cyclomatic complexity are also very much interchangeable in nearly any of measurements we attempted to measure at predict running time amongst members of our data set. Although our results cannot be said to generalize to all types of applications, our set of 21 tasks boasts a high mean and wide standard deviation for initial edges in input graphs, giving us some confidence that our results can generalize to unobserved EFG's both larger and smaller than the ones studied.

In summary

- We found that there is a statistically significant positive correlation between running time, and graph size of the EFG used to generate test cases (post the final reduction phase) for our technique. Thus, we can conclude test case generation instances with longer running times tended to have higher reduction-3 graph sizes. Not only that, but the edge count is quite strong and accurate in predicting running time. ($r = .647$, $p = .002$)

- We found that there is a statistically significant positive correlation between the input EFG graph size and the running time of the HPRT generator. In addition to those with larger sizes after reduction 3, instances with larger initial EFG inputs, have longer running times. ($r = .557$, $p = .031$).

- We found that EFG size and cyclomatic complexity are equally reliable measurements that help one to find among the running times of the generator on a

Table 6.9: Correlations between Graph Size (Edge Count) and Running Time

Variable	Mean	Standard Deviation	$r_{EFStime}$ (Significance)	$r_{HPRTtime}$ (Significance)
HPRT time	957.400	23.74	.714 (.003)	-
EFS time	11.333	27.93	-	.714 (.003)
Edges init	54.43	25.94	.654 (.001)	.627 (.012)
Cyc. Comp. init	47.14	24.92	.585 (.005)	.583 (.022)

given input EFG. There is a positive correlation, which means that higher edge counts and higher cyclomatic complexities predict longer running times. This may not hold as strongly for predicting the running time of the HPRT generator as it might for EFS generator, according to the data we collected.

Summary of RQ3: We conclude that each of the steps of the EventFlowSlicer algortihm reduce the number of edges in the graph, and conclude in accordance with earlier results that edge counts at the start and end of the algorithm's process may be useful to predict how long it will take for the generator to finish, using statistical evidence as our backing. We also conclude that a trend exists among cyclomatic complexity measurements on the same graphs that can help predict this running time just as well.

6.4.5 Threats to Validity

As with all empirical studies there are some threats to validity of this work. First we evaluated our techniques on a limited number of subjects. However, we used subjects both from published research as well as from online forums, therefore we believe that these are represenatitve.

A limitation of the end to end method we provide is that we do not currently allow for the capture of keyboard shortcuts used on the interface, to create model files containing keyboard shortcuts. The old method also did not allow for the capture of these. In addition both generators will interpret input files and generate test case files

containing keyboard shortcuts commands, though they can only capture test cases for a GUITAR based replayer of a very select few keystrokes (namely arrow keys and space bar), and only replay test cases containing those few keystrokes at this time via its replay mechanism.

A certain amount of background knowledge on how to handcode these in EFG and Rules files is necessary in order to properly generate models containing keyboard shortcuts, since we don't support capture using keyboard shortcuts.

After looking at the results, we admit that as compared to the constaraints specified in the HPRT study on HPRT generator, the EventFlowSlicer often requires the specification of more constraints than would be used for the HPRT generator to get the same test cases. However, there is an added benefit: adding more control to test case output, we effectively eliminate the need to constraint the test case generator to only generate tests of a given length. Our test case generator does not accept a length parameter, but in turn we have added more constraints to the base set of constraints defined by the HPRT generator, to give the user added control.

Chapter 7

Conclusions and Future Work

In this thesis, we presented EventFlowSlicer, a tool that extends the HPRT test generation strategy for functional GUI testing by providing a way for a user to generate all test cases that satisfy a particular user goal. EventFlowSlicer helps testers generate GUI test cases for specific goals which can vary based on structural differences, functional differences and for goals that are abstract.

We first performed a feasibility study to show that we can use EventFlowSlicer to generate tests for a small number of tasks. We then performed an empirical study to evaluate it more thoroughly. In that study we see that our generator produces the same test cases as the HPRT method in a fraction of the time. We also find that we can generate tests for a variety of other tasks which have different types of goals. We find that the user goals have on average 38 test cases and as many as 200 which indicates the need for an automated technique.

As for future work we will conduct larger scale studies for both functional testing and human performance testing. We will implement our tool within the CogTool Helper framework and extend the generator to more types of goals.

Bibliography

[1] RTI, "The economic impacts of inadequate infrastructure for software testing," National Institute of Standards & Technology, National Institute of Standards and Technology, Acquisition and Assistance Division, Building 101, Room A1000, Gaithersburg, MD 20899-0001, Technical Report 7007.011, May 2002.

[2] A. Swearngin, M. Cohen, B. John, and R. Bellamy, "Easing the generation of predictive human performance models from legacy systems," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '12, 2012, pp. 2489–2498.

[3] R. Bellamy, B. John, and S. Kogan, "Deploying CogTool: integrating quantitative usability assessment into real-world software development," in *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, May 2011, pp. 691–700. [Online]. Available: http://dx.doi.org/10.1145/1985793.1985890

[4] A. M. Memon and Q. Xie, "Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software," *Software Engineering, IEEE Transactions on*, vol. 31, no. 10, pp. 884–896, 2005.

[5] S. Zhang, H. Lü, and M. D. Ernst, "Automatically Repairing Broken Workflows for Evolving GUI Applications," in *Proceedings of the 2013 International Symposium*

on Software Testing and Analysis, ser. ISSTA 2013. New York, NY, USA: ACM, 2013, pp. 45–55. [Online]. Available: http://dx.doi.org/10.1145/2483760.2483775

[6] A. Swearngin, M. B. Cohen, B. E. John, and R. K. Bellamy, "Human performance regression testing," *Proceedings of the 2013 International Conference on Software Engineering*, pp. 152–161, 2013.

[7] B. N. Nguyen, "Testing gui-based software with undetermined input spaces," Ph.D. dissertation, University of Maryland, College Park, 2013.

[8] A. Bertolino, "Software testing research: Achievements, challenges, and dreams," *Future of Software Engineering*, pp. 85–103, 2007.

[9] F. Gross, G. Fraser, and A. Zeller, "Search-based system testing: High coverage no false alarms," *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, vol. 2012, pp. 67–77, 2012.

[10] A. M. Memon, "An event-flow model of GUI-based applications for testing," *Journal of Software Testing, Verification and Reliability*, vol. 17, pp. 137–157, 2007.

[11] A. M. Memon and Q. Xie, "Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software," *Software Engineering, IEEE Transactions on*, vol. 31, no. 10, pp. 884–896, Oct. 2005.

[12] R. M. L. M. Moreira and A. C. R. Paiva, "Pbgt tool: An integrated modeling and testing environment for pattern-based gui testing," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 863–866. [Online]. Available: http://dx.doi.org/10.1145/2642937.2648618

[13] G. Bae, G. Rothermel, and D.-H. Bae, "On the Relative Strengths of Model-Based and Dynamic Event Extraction-Based GUI Testing Techniques: An Empirical Study," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE, Nov. 2012, pp. 181–190. [Online]. Available: http://dx.doi.org/10.1109/issre.2012.18

[14] X. Yuan, M. Cohen, and A. Memon, "GUI interaction testing: Incorporating event context," *IEEE Transactions on Software Engineering*, vol. 37, no. 4, pp. 559 –574, 2011.

[15] S. Arlt, A. Podelski, I. Banerjee, A. M. Memon, and M. Schaf, *Lightweight Static Analysis for GUI Testing*. IEEE 23rd International Symposium on Software Reliability Engineering, 2012.

[16] S. Carino and J. H. Andrews, "Dynamically Testing GUIs Using Ant Colony Optimization (T)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, Nov. 2015, pp. 138–148. [Online]. Available: http://dx.doi.org/10.1109/ase.2015.70

[17] L. Mariani, M. Pezze, O. Riganelli, and M. Santoro, "AutoBlackTest: Automatic Black-Box Testing of Interactive Applications," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, Apr. 2012, pp. 81–90. [Online]. Available: http://dx.doi.org/10.1109/icst.2012.88

[18] A. Swearngin, "Cogtool-helper: Leveraging gui functional testing tools to generate predictive human performance models: A thesis," Master's thesis, University of Nebraska, Lincoln, 2012.

[19] A. M. Memon, M. E. Pollack, and M. L. Soffa, "Hierarchical gui test case generation using automated planning," *IEEE Trans. Softw. Eng.*, vol. 27, no. 2, pp. 144–155, Feb. 2001.

[20] A. Memon, I. Banerjee, and A. Nagarajan, "Gui ripping: Reverse engineering of graphical user interfaces for testing," *Proceedings of the 10th Working Conference on Reverse Engineering*, pp. 260–269, 2003.

[21] A. M. Memon, "Advances in GUI testing," in *Highly Dependable Software – Advances in Computers*, M. V. Zelkowitz, Ed. Academic Press, 2003, vol. 58, pp. 149–201.

[22] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "Guitar: an innovative tool for automated testing of GUI-driven software," *Automated Software Engineering: An International Journal*, vol. 21, pp. 65–105, 2013.

[23] A. Rodrigues da Silva, "Model-driven engineering: A survey supported by the unified conceptual model," *Computer Languages, Systems & Structures*, Jun. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.cl.2015.06.001

[24] M. Micallef and C. Colombo, "Lessons learnt from using DSLs for automated software testing," in *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*. IEEE, Apr. 2015, pp. 1–6. [Online]. Available: http://dx.doi.org/10.1109/icstw.2015.7107472

[25] T. Monteiro and A. C. R. Paiva, "Pattern Based GUI Testing Modeling Environment," in *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*. IEEE, Mar. 2013, pp. 140–143. [Online]. Available: http://dx.doi.org/10.1109/icstw.2013.24

[26] A. M. P. Grilo, A. C. R. Paiva, and J. a. P. Faria, "Reverse engineering of GUI models for testing," in *5th Iberian Conference on Information Systems and Technologies*. IEEE, Jun. 2010, pp. 1–6. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5556690

[27] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Vancouver, BC*, October 2010.

[28] A. Agarwal and M. Prabaker, "Building on the usability study: Two explorations on how to better understand an interface," in *Human-Computer Interaction. New Trends*. Springer, 2009, pp. 385–394.

[29] (2016, Jun.) Gherkin cucumber/cucumber Wiki Github. [Online]. Available: https://github.com/cucumber/cucumber/wiki/Gherkin

[30] (2015, Oct.) jEdit - Programmer's Text Editor - overview:. [Online]. Available: http://jedit.org/

[31] (2016, Jul.) DrJava. [Online]. Available: http://drjava.org/

[32] A. Memon. (2015, Apr.) TerpOffice 3.0 Artifacts. [Online]. Available: http://www.cs.umd.edu/~atif/TerpOffice/